MIIS REFERENCE MANUAL

VERSION S.MIIS.R-II-2.1

COPYRIGHT © 1981 BY MEDICAL INFORMATION TECHNOLOGY, INC.
255 BENT STREET, CAMBRIDGE, MASSACHUSETTS 02141
(617) 354-3000

THIS INFORMATION IS PROPRIETARY AND SHOULD BE TREATED ACCORDINGLY.

1 Preface....1

2 GENERAL ENVIRONMENT
2.1 Allocation of Computer Memory....2
2.2 Passwords and their Implications....3
2.3 Installing and Accessing MIIS....4
2.4 Interpreter and MIIS as a Control Language....6

3 Commands, Functions, Reserved Words....7

4 Programs - Input, Edit, Delete....10

5 Indirection....15

6 EXPRESSIONS
6.1 Syntax Diagrams....17
6.2 Data Values and Collation Sequence....20
6.3 Data Names....20
6.4 Collation of Data Names....23
6.5 Operators....24

7 INPUT AND OUTPUT
7.1 READ and WRITE Commands....29
7.2 ASSIGN and UNASSIGN Commands....33
7.3 $ASSIGNMENT and $JOB....43
7.4 View Buffer, TRANSFER Command, $WORD and $BYTE....44

8 PROGRAM FLOW AND LOGIC
8.1 GOTO and OVERLAY Commands....47
8.2 DO, CALL and QUIT Commands....49
8.3 Subroutines....53
8.4 IF, ELSE and IF-SWITCH....55
8.5 FOR Command....57

9 DATA ASSIGNMENT AND DETERMINATIONS
9.1 SET and VARIABLES Commands....59
9.2 KILL Command....61
9.3 $UPDATE Function....63
9.4 NAMESPACE Command....64
9.5 STRUCTURED SET Commands....65
9.6 $DEFINE and $NEXT Functions....67
9.7 $GLOBALGET and $G Reserved Word....69

10 STRING MANIPULATION
10.1 String Manipulation Operators....72
10.2 String Manipulation Functions....72

11 MISCELLANEOUS COMMANDS
11.1 XECUTE Command....80
11.2 JOB Command....81
11.3 HALT and HANG Commands....82
11.4 BREAK and GO Commands....83
11.5 Comments in Programs....84

(CONTINUED)

(CONTINUED)

12 MISC. FUNCTIONS AND RESERVED WORDS
12.1 $TIME, $CONSOLE, and $KONSTANT....85
12.2 $X-COORDINATE and $Y-COORDINATE....87
12.3 $IO, $STORAGE, $PASSWORD, $QUEUE, $RESOURCE....88
12.4 $MULTI-PRECISION and $RANDOM....90

13 DATA STORED ON THE DISK
13.1 Physical Nature of the Disk....92
13.2 Physical Storage of Globals and Programs....94
13.3 File Protection: OPEN, CLOSE and $UPDATE....98

14 THE RUN-TIME OPERATING SYSTEM
14.1 Different Job States....101
14.2 Local Symbol Table and $HASH Reserved Word....103
14.3 Program Buffers....104
14.4 Global Buffers and Disk Management....106
14.5 The Timesharing Executive....109

15 Summary of Significant Utility Programs....110

16 Appendix I - ASCII Codes....115

17 Appendix II - Disk Block Structure....116

18 Appendix III - Summary of Commands....120

19 Appendix IV - Console Switches and DDT....123

20 Appendix V - List of Error Messages....125

INDEX....127

OS120781

CHAPTER 1: Preface

This Reference Manual is the definitive description of the features supported by the MIIS language and operating system. The manual is in fact a reference document, rather than a self-teaching mechanism, and while an experienced programmer might find it simple to gain a working knowledge of MIIS with this manual, it has not been organized as a teaching tool.

As a language, MIIS statements have a well-defined syntax that must be carefully followed in order to obtain the anticipated results. These syntaxes are presented in a generalized form throughout the manual, with numerous examples. In order to express these generalized forms, it is necessary to incorporate names for some of the parts that make up the forms. These names are represented conventionally by the symbols summarized below:

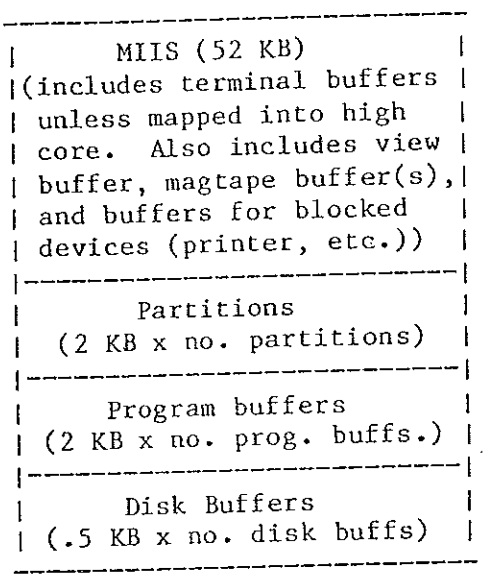
EXP	Any well-formed MIIS expression, as defined in section 6.1, Syntax Diagrams.
VAR	Any well-formed MIIS identifier, as defined in section 6.3, Data Names.
TAG	Any well-formed MIIS program label, as defined in section 4, Programs - Input, Edit, Delete.
PGM	Any well-formed MIIS program name.
NEXP	Expression that evaluates to a number.
LEXP	Expression evaluated as TRUE or FALSE.
NLIT	Numeric literal: one or more digits.
ALIT	Alpha literal: one or more alpha characters.
LHS, RHS	Left-hand-side, Right-hand-side.
n,m,p	Used to indicate positive integers in the various forms.
{ }	Used to surround optional syntax.
< >	Used to surround text describing certain syntax, such as: <Alphanumeric characters>.

Any other text used in any of the schematic syntax definitions is to be interpreted as literal text required in the syntax.

CHAPTER 2: GENERAL ENVIRONMENT

2.1 Allocation of Computer Memory

MIIS uses computer memory as follows:



The MIIS Operating System is a stand-alone time-sharing system that is not subordinate to any other operating system. As illustrated above, MIIS is entirely core-resident, using pre-allocated workspaces. This workspace is allocated at the time each system is configured.

Partitions

The fundamental unit of time-sharing is the "job". Each simultaneous user is assigned an individual workspace called a "partition" which is associated with a job number. Each partition contains three things:

1) The user's local symbol table, which contains data which is accessable only to the individual job.

2) The user's stack, which is used by MIIS both as a scratch-pad for intermediate results, and as a return-address stack for dealing with nested subroutines.

3) The program vector, which contains other information local to the processing of the job.

one grows from one end, one from the other?

The partition stores 2048 bytes, of which roughly 50 are consumed by the job's program vector. Attempts to allow the combined size of the stack and symbol table to exceed the limitations of the remaining space will result in an error.

Programs

MIIS programs are simply stored lines of textual information. This information (MIIS code) is interpreted by MIIS as instructions to the operating system to perform some action. Each user has, in addition to his partition, access to space for storage of a "currently-in-use" program. This space, known as the program buffer pool, consists of a number of 2 KB buffers, each capable of holding exactly one MIIS program. If each user of a MIIS system is executing a different program, then each job is linked to its own program buffer. If multiple users are executing the same program, then those jobs share a single copy of the program by linking to the same program buffer.

The important point here is that while a user's symbol table and stack are local to his job, and that space is accordingly allocated to this purpose on a per-job basis, user programs are accessible to any job because they are stored in a common pool.

Disk Buffers

Disk blocks are also accessible to any job running on a machine, and are stored in a common pool when they are brought into memory for use. This pool is managed by a least-recently-used algorithm so as to maximize the common utilization of frequently accessed disk blocks. The details of this management are outlined in a later section.

2.2 Passwords and their Implications

It is often the case that a given "application" (set of programs and data) is not of sufficient size to warrant the use of an entire computer. In addition, many applications are of a highly volatile nature, requiring frequent changes to their programs. In either case, it is necessary to allow multiple sets of programs and data to share a computer (in the first case for economy, and in the second to allow program changes to be tested without affecting the user's environment).

Sets of programs and data are called "passwords" in MIIS. Each MIIS system is capable of supporting a number of separate passwords. Each password is composed of 4 things:

1) A beginning point and (optionally) an end point which bounds the storage of that password's information on the disk.

2) A program directory block and a global directory block, which are the roots of the B-trees which store that information (this is covered in more detail in a later section).

3) A three-letter code which is used when signing on to tell MIIS which program/data space you wish to access.

4) (Optionally), a MIIS program (written by the user) which is invoked whenever anyone accesses the password in programming mode.

Passwords are set up by a utility program called "PW" which is documented in the MIIS Utilities Manual. That manual also describes a global called ^%PW which stores some external descriptive information about each password.

Password Table

There is a table in the operating system called the password table which stores most of the information described above. A copy of this table is maintained on the disk image of the operating system by the password utilities.

System Password

When MIIS is delivered, there is always one entry already made in the password table. This entry is for password number one, also called the system password. This is where the MIIS utility programs and globals are stored. The names of certain of these programs and globals have a percent-sign (%) as their first character. Such programs and globals are accessible from all passwords (this is the only exception to the inter-password barrier).

2.3 Installing and Accessing MIIS

Loading MIIS

Because MIIS is a stand-alone operating system, it must be installed via a bootstrap procedure. This mechanism varies with the particular hardware configuration (details are provided in the MIIS Utilities Manual), but there are some common points. MIIS is delivered on some transportable magnetic medium (magtape or diskette) and is loaded into core using a hardware bootstrap. The newly loaded system is then copied to the first physical cylinder of the disk, from where it may be bootstrapped into memory whenever necessary.

Patching MIIS

It is sometimes necessary to patch MIIS (modify the operating system) in order to remedy a malfunction. These patches are provided by MEDITECH, usually after telephone consultation, and are implemented using either the

Dynamic Debugging Technique (DDT) capability or the system patch utility (PTS). It is important that the user realize that both the core copy and the disk image of MIIS may be modified. Local demands usually dictate which copy to patch.

Gaining Access to MIIS

In order to sign on to the system, it is necessary to indicate to MIIS which password (set of programs and data) you wish to access, and in which of two modes you wish to make the access.

In the first mode, called "programming mode", you are allowed to interact directly with MIIS. This interaction takes the form of a dialogue in which the user types in MIIS commands, and MIIS reacts to them. In this mode, the user may create and modify programs and access the database directly, without running an existing MIIS program or utility.

In the second mode, called "user mode", the user indicates the name of a MIIS program which he would like to run. This program is typically a menu-selection program, which enables the user to choose among the various options available to him within the application. In this mode, the user is not allowed to interact directly with the MIIS Operating System - all of his activity is under applications program control. It should be noted that the name of the program which represents the "front-end" to the application must have a pound-sign ("#") appended to it (for example, "OPT#" is a popular name for the option reader). This is to prevent the user from accidentally invoking a program which is normally protected by the application.

The procedure for signing on is as follows:

After depressing the break key on a vacant terminal (or sending a CTRL/@), the user is prompted:

```
SYSTEM NAME nn  [The "nn" represents the logical line
PW              number to which the terminal is connected.]
```

The user enters the three-letter code that corresponds to the particular password or database he wishes to access, followed by a carriage-return. If an invalid code is entered, or if a valid code is not entered within 10 seconds, the system aborts the sign-on process. Otherwise, the system makes a further prompt:

```
SYSTEM NAME nn
PW   ID
```

It is here that the user indicates whether he intends to write programs or to run an application. To run an application, he enters the name of an applications program as outlined above, WITHOUT THE "#". To enter programming mode, the user enters a different code here. In either case, the entry is followed by a carriage-return. Once again, the user has 10 seconds in which to enter a valid "ID", or the sign-on process is aborted.

It should be noted that it is possible to make some ports (user terminal

connections) "non-programmable", in order to increase the security of a database. There is a MIIS Utility Program called "PRG#" which resides in the System Password which maintains a table of port programmablilty. More information is available in the MIIS Utilities Manual.

2.4 Interpreter and MIIS as a Control Language

Once the user has signed on in programming mode, anything he types will be considered an attempt to enter input to the MIIS interpreter. Whenever MIIS is not in the process of executing commands, it is waiting for such input and indicates its readiness by advancing to a new line and typing a question mark ("?") at the left-hand margin.

After a question mark, all text entered by the programmer until the return key is depressed constitutes a statement, which may be made up of one or more MIIS commands. If the input text is not a legal MIIS statement, a message describing the nature of the error is printed and the process is aborted. If the input text is a legal MIIS statement, it is executed immediately. In either case, MIIS then advances to a new line to await another statement.

This mode of operation (where commands are input, interpreted, and executed in an apparently single operation) is called direct mode. Almost all MIIS commands may be interpreted like this outside the context of a program.

MIIS programs are made up of stored lines of such statements. Each line of a MIIS program is entered with the following format:

{tag}<tab><statement>

These program lines are entered into MIIS just as though the user was entering code for direct mode execution. The presence of the <tab> character (CTRL/I, or on some terminals, a separate key) tells MIIS that the user intends the current input to be stored in the user's current program buffer rather than executed immediately. The {optional} tag which preceeds the <tab> character is used to identify the line from inside the program. The line's physical location within the program (sequence of many such lines) is determined by the value of the user's line pointer at the time the line was entered. This concept is explained further in section 4, Programs - Input, Edit, Delete.

CHAPTER 3: Commands, Functions, Reserved Words

There are three types of constructs that form an integral part of the MIIS language: commands, functions, and reserved words. Each command, function, and reserved word has a name. The form that the name takes depends on the particular construct:

Construct	Form of Name
Command	name
Function	$name(...)
Reserved Word	$name

IMPORTANT: In each construct, the name itself may be abbreviated by its first letter.

Commands

Commands are direct instructions to the MIIS interpreter to perform some action.

Each command consists of a command name, possibly followed by a condition under which the command is to be performed, possibly followed by a set of arguments that further specify the action of the command.

All commands except FOR, IF, and ELSE may be conditionalized by appending ":EXPRESSION" to the name of the command. For example:

```
SET:A>1 A=1  assigns the value "1" to the variable
             A if and only if the current value of
             A is greater than 1.
```

Commands may take zero, one, or more than one arguments, and a program (or direct mode) line of code may contain several commands. If a line of code contains more than one command, each command is separated from its neighbor by a single space character, with the following exceptions:

1) Any instance where an IF, ELSE, NAMESPACE, CLOSE, KILL, LOAD, PRINT or FILE command occurs without an argument requires TWO spaces preceding another command on the same line.

2) Any instance where a BREAK or HALT command occurs requires TWO spaces preceding another command on the same line.

Repeated applications of the same command, each containing one or more arguments, may be condensed into a single application of the command with each argument or argument list consolidated into one list. In other words:

```
COMMAND ARG1,ARG2 COMMAND ARG3 is exactly equivalent to
COMMAND ARG1,ARG2,ARG3
```

Note that any conditionalization applied to a command extends over the entire list of arguments that follow the command. For example:

```
NAMESPACE:LEXP 1=^TEMP,2=^ALPHA
```

If LEXP evaluates to a true condition, the entire list of namespace associations will be made. Otherwise, none of the namespaces will be created.

Functions

Functions are predefined procedures that offer certain useful capabilities which are not provided by the set of commands and operators.

Each function has a name, and the action of the function is specified by a list of arguments. All MIIS functions return a value when evaluated and may therefore be used as atoms in constructing MIIS expressions.

The general form of a MIIS function is:

```
$name(arg1,arg2,...,argn)
```

The number and nature of the arguments of course depend on the particular function. In several cases, the action of the function may depend on the number of arguments specified.

Functions are separated into several classes:

Class	Functions in Class
String Manipulation	$CHARACTER, $EXTRACT, $FIND, $HASHCODE, $INVERSE, $LENGTH, and $PIECE.
Data Definition	$DEFINE, $GLOBALGET, $NEXT, and $UPDATE.
Memory Inspection and Editing	$BYTE and $WORD.
Miscellaneous	$ASSIGN, $JOB, $MULTIPRECISION, $RANDOM, $TEXT, and $XCLUSIVEOR.

Reserved Words

Reserved words are essentially system variables that express some state of the MIIS operating system.

Each reserved word has a name. The general format of a reserved word is:

$name

Reserved words, like functions, have values and may be used as atoms in construction of MIIS expressions.

Reserved words fall into two categories: status information over an entire computer, and status information local to a particular job or device.

Class	Reserved Words in Class
Entire computer	$CONSOLE, $KONSTANT, $TIME.
Particular job or device	$ASSIGNMENT, $G, $IO, $JOB, $PASSWORD, $QUEUE, $RESOURCE, $STORAGE, $XCOORDINATE, and $YCOORDINATE.

limited to 3 or 4 significant characters?

Three of the reserved words ($PASSWORD, $QUEUE, and $RESOURCE) may be explicitly SET (established) by the user in a program. The other reserved words are managed by the MIIS operating system.

CHAPTER 4: Programs - Input, Edit, Delete

As noted in section 2.4, Interpreter and MIIS as a Control Language, MIIS programs are simply stored lines of MIIS statements. Each line of text (code) in the program consists of an optional label, a tab character, and a collection of MIIS statements:

{TAG}<tab><MIIS statements>.

Programs which are visible to a MIIS programmer (because they are being typed in, edited, viewed, or executed) are stored in a program buffer which is attached to the programmer's partition. Programs which are stored for future use are stored on the computer's long-term storage device (usually the disk) in the password area in which they were filed.

There are five commands and one function associated with MIIS programs: the PRINT, ERASE, and MOVE commands and the $TEXT function refer to program text stored in a program buffer. The LOAD and FILE commands transfer programs between the program buffers and the long-term storage (disk).

Physical Limitations on Programs

Program buffers are capable of storing 2047 bytes of program text. This byte-count includes all characters in the program, including the tab characters, the carriage-returns and the tag offsets. Attempts to exceed this limit result in an error.

There is no limit to the number of characters in a program line.

Program Labels (TAGs)

The PRINT, ERASE, MOVE and $TEXT statements may all refer to specific lines of a program. Lines are identified by reference to a program label, optionally plus a positive offset. Program labels appear as a string of characters before the tab character in a program line. Program labels have the following format:

{%}<alphanumerics>{#}

There is no restriction on the size of a program label. The pound-sign (#) which optionally appears at the end of the label is used only on labels which identify the first lines of programs, and indicates that the program may be executed directly on sign-on in user mode (see 2.3, Installing and Accessing MIIS).

There is in addition an implicit NULL label which refers to the first line of the program so that, for example, line "+0" refers to the first line of the program, "+1" to the second, and so on.

Typing Programs In (Program Line Pointer)

One way to get program lines into a job's program buffer is to type them in from a terminal device (see 2.4, Interpreter and MIIS as a Control Language). Input from a terminal in response to the MIIS "?" prompt is interpreted as a program line if the input looks like:

{<alphanumeric text{#}>}<tab character>{<anything>}

When MIIS receives a program line, it stores it in the program buffer after the line currently indicated by the PROGRAM LINE POINTER. This pointer is maintained by MIIS, and is changed whenever a PRINT, ERASE, or MOVE command is executed, and is incremented by one whenever a program line is entered into the buffer. When a program is first loaded into the buffer, the line pointer is in an indeterminate state, and should be explicitly adjusted. When a program buffer is ERASEd, the line pointer is in a state that allows the first-entered program line to become the first line of the program.

PRINT Command

The PRINT command allows the contents of a partition's program buffer to be output to a peripheral device. The output device is the job's current I/O device. The form of the command is:

PRINT {{TAG}:{TAG}}

For example:

Command	Meaning
PRINT	Output the entire program.
PRINT A:	Output lines from A through the end.
PRINT :B	Output lines from the beginning through B.
PRINT A:B	Output lines from A through B.

The PRINT command outputs a line that consists of a single space character as the last line of its output. This line (which is invisible on a CRT or printer) is used so that the end of the program may be recognized by an intelligent peripheral device.

The PRINT command leaves the program line pointer pointing to the last line printed.

ERASE Command

The ERASE command is used to remove program lines from the program buffer attached to a partition. The general format is:

```
ERASE {TAG:TAG}
```

For example:

Command	Meaning
ERASE	Erase all program lines from the job's program buffer.
ERASE A:B	Erase all lines between line A and line B, inclusive.

The ERASE command leaves the program line pointer at the line immediately preceeding the first line erased. If the first line of the program is erased, the program line pointer points to a conceptual "zero'th" line.

MOVE Command

The MOVE command is really three commands in one, depending on the nature and number of the arguments to which it is applied. In the first instance, there are zero arguments:

```
MOVE
```

and the effect of the command is to move the program line pointer to before the first program line, thereby enabling the programmer to enter a new first program line.

In the second instance, the argument is a program tag:

```
MOVE TAG
```

and the effect is to move the program line pointer so that it points to the line specified by the argument.

In the third instance, the argument is a program tag and the text of a program line:

```
MOVE TAG:<program line>
```

The effect of this command is to insert the <program line> into the program after the specified tag. The program line pointer is left pointing to the new program line.

$TEXT Function

The $TEXT function has a program tag as its argument and returns the text of the program line specified by that tag as its value. If the line specified by TAG does not exist, the nil value is returned.

$TEXT(TAG)=<program line named TAG>

For example:

Example	Meaning
MOVE TAG:$TEXT(TAG)	Create a line identical to line TAG, stored immediately after line TAG.
$TEXT(+0)	The text of the first line of the currently loaded program.

LOAD Command

The LOAD command is used to get a program into a program buffer.

If the source of the program is some peripheral device (such as a CRT or a magnetic tape) specified as the current I/O device at the time the LOAD command was issued, then the LOAD command has no arguments. In this event, the load is terminated by a line containing a single space character, such as is output by the PRINT command as its last line.

Issuing the LOAD command with no arguments causes the user's program buffer to be erased before the new program is loaded. Note that this implies that any such command must be executed from the user's stack, rather than from a loaded program (i.e., the LOAD must either be XECUTEd or be a direct-mode command).

If instead of a peripheral device, the source of the program is the password's program directory stored on disk, then the argument of the LOAD command is the name of the program:

```
LOAD PGM
```

Attempts to load a non-existant program from the program directory will result in an error, and the previously-loaded program will not be erased in that event.

Information on the dynamics of program loading (i.e., management of the system's program buffers) is given in section 14.3, Program Buffers.

FILE Command

The FILE command serves two purposes: to copy programs from a program buffer to the password's program directory, and to delete programs from a password's program directory.

In the first instance, the FILE command is given with no arguments. The program is filed in the program directory of the password to which the job is attached at the time the FILE command is given, under the name given as the tag of the first line of the program. For example, if the following program was loaded, and a FILE command was issued, the program would be filed as "MFP" in the current password:

```
MFP     ;MY FIRST PROGRAM
        READ !,"ENTER A NUMBER: ",A WRITE !,"THE NUMBER IS: ",A QUIT
```

The first line of the program must have a tag which is a valid program name, or else executing the FILE command will give an error. Valid program names are:

Any valid TAG, or
Any TAG with a "#" appended, or
Either of the above preceeded by a "%".

In the second instance of the FILE command, the name of the program to be deleted is specified as an argument to the FILE command, and to be sure there is no mistake, the text ":X" is appended to the name of the program:

```
FILE PGM:X
```

Any attempts to FILE a program with a name beginning with a percent-sign (%) without the SYSTEM switch being raised on the (possibly virtual) computer console will result in an error. This is to protect against accidental or malicious alteration of the system utility programs.

CHAPTER 5: Indirection

Most "machine" languages have a mode of addressing called "indirect addressing", as opposed to "direct addressing".

Direct addressing traditionally means that in order to access the contents of a particular memory location, a program must address that location as part of the instruction itself.

Indirect addressing involves an intermediate step: rather than specifying the location itself, a different location is identified which in turn "points" to the "real" location. This mode of addressing is useful when accessing a series of values stored in a table, because a single instruction may be used in the loop, and the value of the pointer may be changed as necessary.

There is a roughly analogous concept of indirection in MIIS. Because MIIS does not deal with absolute locations, and does deal only with data names, the concept is translated accordingly.

All commands except the FOR command may be expressed indirectly. Recall from section 3, Commands, Functions, Reserved Words that every command takes zero, one, or more than one argument, and that commands with more than one argument require that each argument be part of a list:

No arguments:	COMMAND	example: KILL
One argument:	COMMAND ARG	example: KILL A
>1 arguments:	COMMAND ARG,ARG	example: KILL A,B

Each argument in the argument list may be specified indirectly, regardless of whether the neighboring arguments are so specified. The indirection operator is the underscore ("_"), which on some terminals is represented as a back-arrow. The underscore must occur as the first character of the argument, and causes all atoms in the expression that forms the argument to be EVALUATED as the MIIS interpreter decodes the argument.

For example, if A="B", B=1, C="^A", then:

INDIRECT CONSTRUCTION	MEANING
WRITE _A	"A" is evaluated to yield the string "B". The command is then equivalent to "WRITE B", which outputs "1".
SET _C_"="_A	The 2nd and 3rd occurances of the underscore indicate concatenation (see section 6.5, Operators). After evaluating the atoms and concatenating the results, the interpreter performs "SET ^A=B".
SET _C_"=A"	Equivalent to "SET ^A=A". (Compare with the previous example.)

SET _A,_C,A	Equivalent to "SET B,^A,A" (Note that the third argument ("A") is not specified indirectly).
IF _"$D("_C_","_A_")"	Equivalent to "IF $D(^A,B)".
WRITE _"A:"_B_"R"_(B+5)	Equivalent to "WRITE A:1R6"

There is a useful trick for developing and verifying the accuracy of the sometimes obscure-looking indirect constructions. In the last example given above:

```
WRITE _"A:"_B_"R"_(B+5)
```

note that the statement:

```
WRITE "A:"_B_"R"_(B+5)
```

yields the string "A:1R6". This is of course because the direct-mode write statement forces MIIS to display the results of the successive evaluation and concatenation of the atoms in the expression, thereby indicating what command will result from the indirection.

CHAPTER 6: EXPRESSIONS

6.1 Syntax Diagrams

The syntax diagrams on the following pages illustrate the structure of well-formed expressions, and give definitive meaning to several terms we have been and will be using:

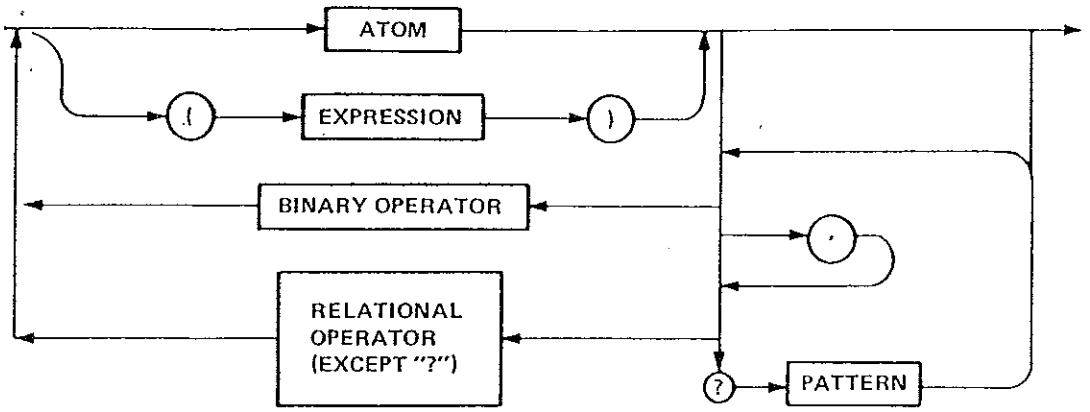
EXPRESSION
ATOM
IDENTIFIER
NAME
PREFIX
LITERAL
PATTERN
RELATIONAL OPERATOR
BINARY OPERATOR

The syntax diagrams graphically show how legal MIIS expressions may be constructed. Their meanings (the effect of interpreting these expressions) are explained in the next few sections.

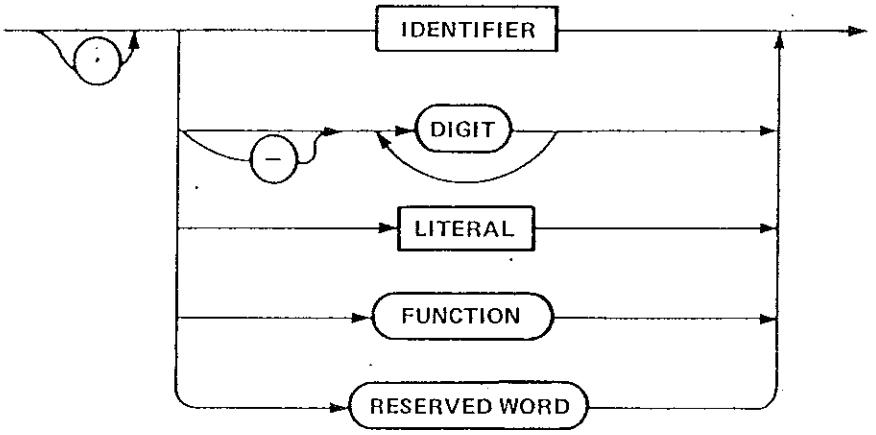
SYNTAX DIAGRAM NUMBER ONE

WELL-FORMED EXPRESSIONS

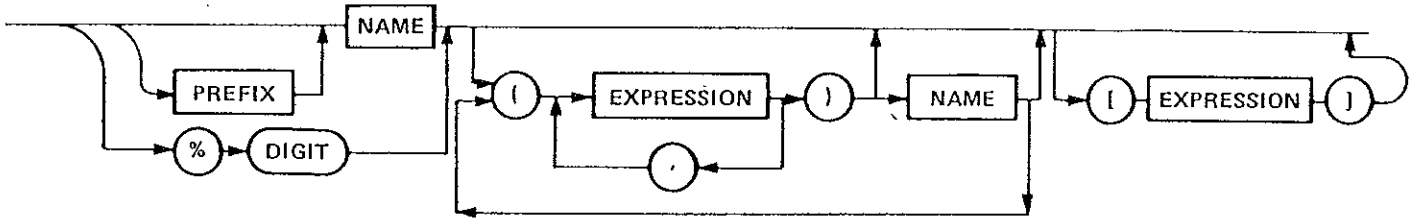
EXPRESSION:



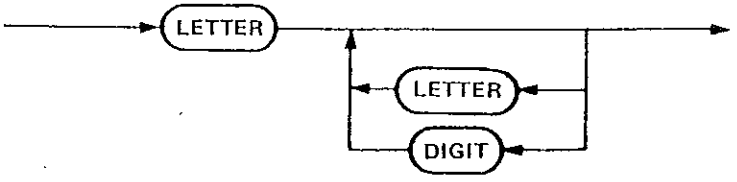
ATOM:



IDENTIFIER:

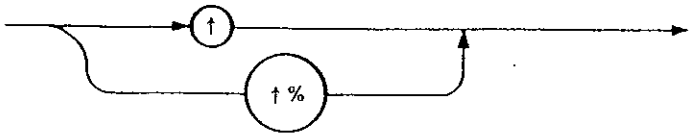


NAME:

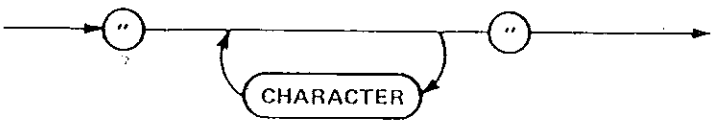


SYNTAX DIAGRAM NUMBER TWO

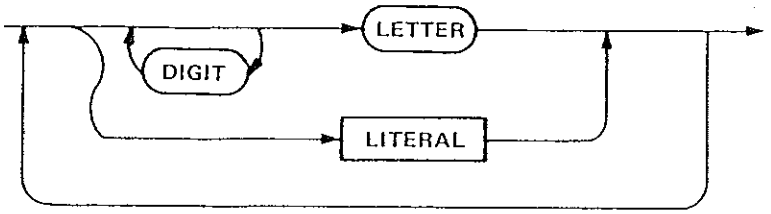
PREFIX:



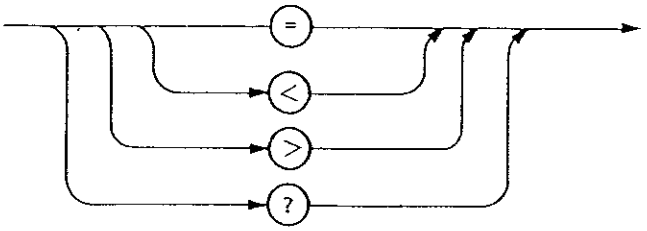
LITERAL:



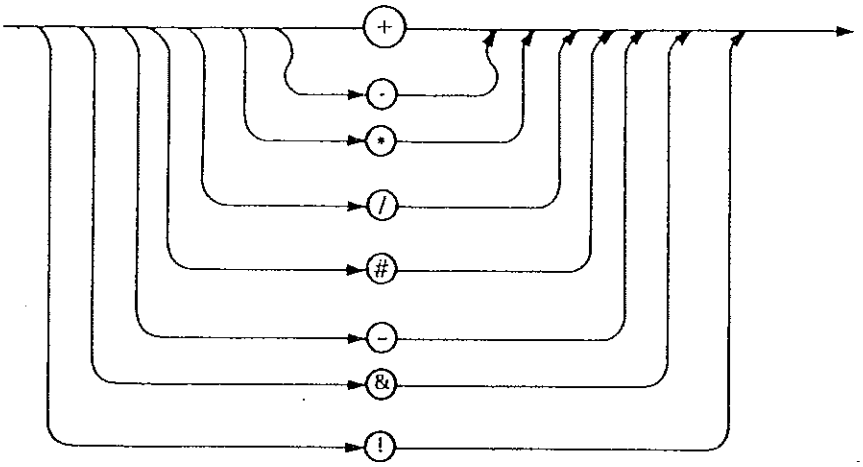
PATTERN:



RELATIONAL OPERATOR:



BINARY OPERATOR:



KEY:

DEFINED

UNDEFINED (DIGIT, LETTER, CHARACTER, FUNCTION, RESERVED WORD)

(·) = LITERAL CHARACTER

6.2 Data Values and Collation Sequence

All MIIS data values are strings formed from the full set of 128 ASCII characters.

Appendix I contains a table of all legal MIIS characters. It is important to realize that MIIS actually stores these values in 8-bit bytes, using the high-order (128) bit to offset the first 32 (0-31) characters. For example, ASCII NULL ($C(0)) is internally represented by a byte with the value of 128 decimal. ASCII SPACE, and all characters following it in sequence (through RUBOUT) are represented internally by bytes with values exactly equal to the ASCII codes.

These discrepancies from the actual ASCII sequencing is significant when interfacing with a non-MIIS technology, and when performing relational operations in MIIS. The MIIS collation sequence is:

SPACE < (ascii sequence) < RUBOUT < NULL < (ascii sequence) < US

(NOTE: "NULL" in the above sequence refers to the ASCII NULL character, as opposed to the NIL string, which contains exactly zero characters).

6.3 Data Names

MIIS data may be categorized in four ways:

local, scalar	global, scalar
local, structured	global, structured

In the above table, the term "local" means that the data is accessible only to the particular job which created it. In addition, any local data established by a job is deleted when the job is signed off (HALTed). Local data is stored in the job's local symbol table, which is a 2 KB partition assigned to the job when it signs on (see 2.1, Allocation of Computer Memory).

The term "global" means that the data is accessible to any job that is signed on to the same password as the job that created the data. In addition, %-globals, which reside in the system password, are accessible to jobs signed on to any password on the computer. Global data is permanent in nature, and is not deleted when the job is signed off. Global data is stored on the computer's mass-storage device (usually a disk, occasionally a diskette).

All data in MIIS, regardless of its type, is represented as an ordered pair:

(NAME,VALUE)

In other words, all stored data has a name, and all named data has a value. The maximum length of any MIIS (name,value) pair is 255 characters.

MIIS data values are composed of character strings made up of ASCII characters, as discussed in section 6.2, Data Values and Collation Sequence.

MIIS data names are either scalar names or multi-dimensional vector names that represent a data structure. Scalar names are represented by alpha-numeric symbols, where the first character of the name is alphabetic. Vector (array) names are represented by scalar names followed immediately by a parenthetically-enclosed subscript list, with the subscripts separated by commas. Each subscript may be any valid MIIS expression that evaluates to any character string.

These subscripted arrays may be used in a natural way to represent tree-structured databases, with each subscript identifying a path (branch) out of a particular node that was identified by the previous subscripts in the name.

References to undefined data (either local or global) DO NOT generate an error, but rather return a <nil> value.

Examples of legal MIIS names are:

A	A123A	LONGvariableNAME
X1	X1(1)	X1("X")
ARRAY(1,-2,"XYZ")	ARRAY("A","B",1,2)	ARRAY("+","~","A","B")

There are four important exceptions to the data-naming rules presented above:

Global Data Names
------ ---- -----

Global data names always begin with an up-arrow (^) to distinguish them from local data names. The data names given in the above list of legal MIIS names could all be preceded by an up-arrow, and would be transformed from local data to global data.

Data names may also begin with an up-arrow-percent (^%), to distinguish them from "ordinary" global names. We call such globals "%-globals". All %-globals are stored in the system password (see section 2.2, Passwords and their Implications), but may be SET, KILLED, and read from any password for which SET $PASSWORD (see section 12.3, $IO, $STORAGE, $PASSWORD, $QUEUE, $RESOURCE) is enabled.

The prefixes "^" and "^%" are themselves legal global variable names.

Percent-variables

Variables created by the operating system to contain the values of arguments passed to subroutines are named as follows:

%<digit>

where the digit is any integer 0-9. Percent-variables may be used in any way other variables are used EXCEPT that they may not be used as NAMESPACE references.

Packed Data

Data names may also be expressed as:

<data name>[nexp] (for example: ^A(2)[2] or A[A+1] or %1[2])

This syntax enables more than one logical piece of data to be stored in the same physical "node" or location. The individual fields are separated by rubout characters, and are identified by the expression inside the square brackets. The first (0'th) field contains an integer that indicates the number of fields present in the data, and is maintained by the operating system. Each remaining field may contain any data value EXCEPT A RUBOUT. Fields are manipulated as follows:

SET VAR	Initialize VAR for use with packed data.
SET VAR[NEXP]=exp	Establish a new value in the NEXP'th field of VAR, and if VAR[0]<NEXP, set VAR[0]=NEXP. Note that SET VAR[0]=val is illegal.
KILL VAR[NEXP]	Remove the data AND DELIMITER stored in the NEXP'th field, and decrement VAR[0] by one. Note that KILL VAR[0] is illegal.

Subfile Notation

There is an alternative method of specifying MIIS data names from that specified above.

This alternative ("subfile") notation makes explicit the relationships between "bound names" (the variables) and "unbound names" (the subfile names) which exist in many data files. The syntax allows unbound names which look like scalar names (start with a letter and contain only letters and numbers) to appear outside the subscript list, provided that any additional parts of the data name begin a new subscript list. For example:

Traditional Name	Alternative Notation
A(I,"CHARGE")	A(I)CHARGE
A(I,"CHARGE",J,"INVOICE",K)	A(I)CHARGE(J)INVOICE(K)
A("ALPHA","ALPHA",1)	A("ALPHA")ALPHA(1)

This method of presenting data names is often more concise and legible than the explicit list of subscripts.

6.4 Collation of Data Names

Data names have a well-ordered collation (sorting) sequence. The rules for comparing two data names are as follows:

1) If either name exists in "subfile notation" (see section 6.3, Data Names), transform the name to normal, parenthetically-enclosed subscripts.

2) If either name contains subscripts that are not fully evaluated (ie, are not literals), evaluate these subscripts and replace them with their values.

3) In order to resolve the ambiguity between alphabetic order and numeric order:

   If either name contains subscripts which start with a number (like "123" or "123ABC" or "-123" or "-123ABC"), find the subscript which contains the largest number of leading digits. Add leading zeros to all other subscripts which start with a number until each such subscript has the same number of leading digits (not including any minus-sign). EXCEPTION: Do not pad any subscript to more than 14 leading digits.

4) Compare the two names as they appear before the subscript list (if any). Make this comparison character-by-character using the collation sequence presented in section 6.2, Data Values and Collation Sequence. If the names are not equal, stop.

   Otherwise, compare each subscript in turn with its corresponding subscript in the other name. Make this comparison character-by-character using the collation sequence presented in section 6.2, Data Values and Collation Sequence ***EXCEPT*** that the sense of numeric comparisons must be reversed if comparing two subscripts which begin with negative numbers.

   After each such comparison, stop if the subscripts are not equal. If one name runs out of subscripts, the other is the larger. Otherwise, the names are equal.

For example, the following names are in collation sequence, where "rubout" means a rubout ($C(127)) character and "tab" means a tab ($C(9)) character:

^%A < ^%AA < ^A < ^A(1) < ^A(2) < ^A(11) < ^A("A") < ^A("a") < ^A(rubout).

Also,

^A("") < ^B < ^B("") < ^B("-999A") < ^B("-999AA") < ^B("-99AA").

Also,

^A(-1,"") < ^A(-1,"A") < ^A(-1,"A","A") < ^A(-1,"B") < ^A(0) < ^A(tab).

6.5 Operators

In the last two sections, we have seen the forms taken by MIIS data values and names. In this section we will see how these data may be connected with MIIS operators.

Expressions which are formed by the rules given in the syntax graphs in section 6.1, Syntax Diagrams, are evaluated without any operator precedence. In other words, values are formed with a strict left-to-right precedence, except that parenthetically-enclosed sub-expressions are evaluated first. This is an important point, and represents a departure from most computer languages.

Integer Operators

There are five binary arithmetic operators, all of which operate only on integers. MIIS converts non-integers to integers as required by these operators. The non-integer is converted starting with the left-most character and continuing until a non-digit is encountered (except that a "-" may be the very first character of an integer). For example, "12+0" is 12, "-12+0" is -12, "12A+0" is 12, and "A12+0" is **ZERO**. These data conversions take place locally in the arithmetic operation, and do not affect any stored or named data.

+	Addition
-	Subtraction
*	Multiplication
/	Integer Division (2/3=0; 3/3=1; 4/3=1; 5/3=1; 6/3=2)
#	Modulus (remainder on division) (2#3=2; 3#3=0; 4#3=1; 5#3=2; 6#3=0)

Relational Operators

There are four relational operators in MIIS:

- < Less Than
- > Greater Than
- = Equals
- ? Pattern Matching

These operators return a TRUE value or a FALSE value. A false value is equal to the NIL string (exactly zero characters) and a true value is equal to a single rubout ($C(127)) character. Therefore:

Expression	Value
1=1	True ($C(127))
1=1=$C(127)	True ($C(127))
1=1=$C(127)=$C(127)	True ($C(127)) (et cetera ...)
1=2<3	True (because (1=2)=NIL, and NIL<3)
"ABC123"?3A1NN	True

The truth-sense of any of the relational operators may be reversed by preceeding the operator with a NOT operator (apostrophe: ′). For example:

Expression	Value
1+1′=3	True, because 2 does not equal 3.
1+1+1′?2N	True, because 3?1N.
A+A′=2*A	True, because A+A′=2 is either NIL or RUBOUT. In either case, multiplying by A gives zero, which is non-NIL. The expression is therefore true, contrary to what you learned in algebra.

Pattern Matching Operator

Although the pattern-matching operator alluded to above ("?") is properly a relational operator (because it returns a TRUE/FALSE value), it has no arithmetic corollary and may therefore require some additional explanation. The left-hand-side (LHS) of the operator may be any MIIS expression, which will be tested for conforming to the pattern specified by the syntax on the right-hand-side (RHS) of the operator. The patterns are specified by code letters that correspond to subsets of the ASCII character set, or by literal text which must be exactly matched. The code letters and their meanings are as follows:

Code	Meaning
A	Alphabetic characters (upper and lower case)
N	Numeric characters 0-9
P	Punctuation characters (not letters and not numbers)
M	Alphabetic or Numeric
T	Numeric or Punctuation
Q	Alphabetic or Punctuation
U	Any ASCII character

Patterns are built up of "parts". Each part is one of the following:

1) A code letter, which will match ANY number of characters on the LHS which fall into the ASCII subset represented by that code letter. Note that this includes ZERO characters on the LHS. Matching stops when a character on the LHS is not matched by the pattern part.

2) A number followed by a code letter, which will match exactly that number of characters on the LHS which fall into the ASCII subset represented by that code letter. Matching stops when the specified number of characters is matched.

3) A string enclosed in "quotation marks", which will match exactly that string on the LHS. Matching stops when the string is exhausted.

These parts are concatenated to form a pattern, which is matched from left-to-right. As matching stops on each part, the next part is matched until the entire pattern has been verified. For example:

String	Pattern	Explanation
354-3000	?3N"-"4N	An American telephone number always looks like three digits, followed by a hyphen, followed by four digits.
$8429.60	?"$"1NN"."2N	American money has a leading dollar-sign, at least one digit, a decimal point, and two digits. Note that "$"N"."2N would match "$.60".
ABC	?1AU or "A"1AU or "AB"PN1AAU or "ABC"	Any of these patterns match the string "ABC".

Boolean Operators

There are two boolean operators in MIIS:

& Minimum
! Maximum

These operate between any two MIIS expressions, and return the value which is the minimum or maximum value of the two. For example:

Expression	Value
1&2	=1, because 1<2.
1!2	=2, because 2>1.
1!5&10	=5, because 1<5<10 (1!5=5; 5&10=5).
"A"!"M"&"Z"	="M", because "A"<"M"<"Z".
A=1&1+1	=2 if A=1, because (RUBOUT&1=1)+1=2.
	=1 if A'=1, because (NIL&1=NIL)+1=1.

This last example indicates that another meaning may be attached to the minimum/maximum operators: and/or. Note that since relational operators return either NIL or RUBOUT as their values, these may be combined with minimum and maximum operators in the same way as a logical and/or would work:

Logical	Physical	Result
TRUE&TRUE	RUBOUT&RUBOUT	RUBOUT (TRUE)
TRUE&FALSE	RUBOUT&NIL	NIL (FALSE)
FALSE&TRUE	NIL&RUBOUT	NIL (FALSE)
FALSE&FALSE	NIL&NIL	NIL (FALSE)
TRUE!TRUE	RUBOUT!RUBOUT	RUBOUT (TRUE)
TRUE!FALSE	RUBOUT!NIL	RUBOUT (TRUE)
FALSE!TRUE	NIL!RUBOUT	RUBOUT (TRUE)
FALSE!FALSE	NIL!NIL	NIL (FALSE)

Unary Not

The unary not operator (apostrophe: ') that we encountered earlier in this section may also be used to "negate" the value of any MIIS expression. The rules are as follows:

Expression	Value
'NIL	RUBOUT
'(non-NIL value)	NIL

Note that ''1 is equal to '('1). The not-operators do not cancel each other, but rather are evaluated with the right-most apostrophe first.

Concatenation Operator

In section 5, Indirection, we saw indirect expressions constructed by connecting several parts of the expression with the MIIS concatenation operator. This operator ("_", an underscore on most terminals, and a back-arrow on some) has the effect of combining the value on the left-hand-side with the value on the right-hand-side. For example:

Expression	Value
"HELLO"_"THERE"	"HELLOTHERE"
"HELLO"_" "_"THERE"	"HELLO THERE"
"PRINT DATE: "_PDT	"PRINT DATE: MARCH 17, 1980" if PDT="MARCH 17, 1980"

For historical reasons, a period (".") may be used instead of an underscore as the concatenation operator. The two are functionally identical.

CHAPTER 7: INPUT AND OUTPUT

7.1 READ and WRITE Commands

READ Command

The READ command is used to accept data from a peripheral device (a CRT, magtape, diskette unit, etc.). The particular device is always the current I/O device as selected by the ASSIGN command (see 7.2, ASSIGN and UNASSIGN Commands).

The READ command takes one or more arguments as follows:

READ arglist

Each argument in the list is one of the following, where the symbol "VAR" represents any legal MIIS identifier except a packed-data name.

Argument	Meaning
VAR	Accept data from the current I/O device. When the ASCII character representing the ENTER KEY (see 7.2, ASSIGN and UNASSIGN Commands) is received, move the input data into the specified variable.
*VAR	Accept a single character from the current I/O device. Return the ASCII code that represents the input character in the specified variable.
VAR:EXP	Accept data from the current I/O device, terminated by the enter key. Evaluate EXP as a number. If the enter key is hit within EXP seconds, store the input data in the specified variable and set the IF-SWITCH false. If the enter key is not hit within EXP seconds, terminate the READ and set the IF-SWITCH true. In that event, any characters entered between the initiation of the READ and the time-out are moved into the specified variable.
*VAR:EXP	Accept a single character from the current I/O device. Evaluate EXP as a number.

	If a character is input within EXP seconds, store the ASCII code representing that character in the specified variable, and set the IF-SWITCH false. If no character is entered within EXP seconds, terminate the READ and set the IF-SWITCH true.
Literal text enclosed in "quotes"	Output the literal text to the current I/O device.
Carriage control instruction	There are three: `#` Output a page-feed sequence as established by the form-feed characteristics table maintained by the "CF" utility program. `!` Output a CR/LF combination as established by the CR/LF characteristics table maintained by the "CF" utility program. `?EXP` Evaluate EXP as a number. By printing spaces, tab to that position on the current line based on the $XCOORDINATE reserved word. Note that characters typed as input to the computer are not tallied in maintaining $XCOORDINATE.

For example:

Example	Meaning
`R !,?5,"PROMPT: ",A`	Write a CR/LF, print the string "PROMPT: " after the 5'th position, and input data into the variable "A".
`R *^A(1,1):5`	Input a single ASCII code into the global variable ^A(1,1). If no input in 5 seconds, time-out and set the IF-SWITCH true.

WRITE Command

The WRITE command is used to output data to a peripheral device. The particular device is always the current I/O device as selected by the ASSIGN command (see section 7.2, ASSIGN and UNASSIGN Commands).

The WRITE command takes either zero arguments or a list of arguments:

```
    WRITE
or, WRITE arglist
```

When zero arguments are present, the effect of the WRITE command is to output all currently defined local symbols and their values to the I/O device.

Where arguments are present, each argument in the list is one of the following:

Argument	Meaning
EXP	Evaluate the expression, and print the value.
*EXP	Evaluate the expression as a number, and print the single character which has that number as its ASCII code.
EXP:n	Evaluate the expression, print "n" spaces followed by the value of the expression.
EXP:{n}Lm	Evaluate the expression, and {optionally, print "n" spaces}. Then print the value of the expression left-justified in a field of width "m". If the length of the value of the expression exceeds "m", overflow the field rather than truncate the value.
EXP:{n}Tm	Evaluate the expression, and {optionally, print "n" spaces}. Then print the value of the expression left-justified in a field of width "m". If the length of the value of the expression exceeds "m", truncate the value to fit the field.
EXP:{n}Rm	Evaluate the expression, and {optionally, print "n" spaces}. Then print the value of the expression right-justified in a field of width "m". If the length of the value of the expression exceeds "m", overflow the field rather than truncate the value.

EXP:{n}Rm.p

Evaluate the expression, and {optionally, print "n" spaces}. Then print the value of the expression right-justified in a field of width "m+p+1", inserting a period before printing the last "p" characters of the value of the expression. If the length of the value of the expression exceeds "m+p+1", overflow the field rather than truncate the value.

If EXP is <nil>, output "n+m" spaces, a period, and "p" zeros.

EXP:{n}Dm

Evaluate the expression as a number, and {optionally, print "n" spaces}. Transform the value of the expression into a MM/DD/YY date by interpreting the value as the number of days since March 1, 1800 (see section 12.1, $TIME, $CONSOLE, and $KONSTANT). Print the date left-justified in a field of width "m". If the length of the date exceeds "m", truncate the date rather than overflow the field.

Carriage control instruction

There are three:

Output a page-feed sequence as established by the form-feed characteristics table maintained by the "CF" utility program.

! Output a CR/LF combination as established by the CR/LF characteristics table maintained by the "CF" utility program.

?EXP Evaluate EXP as a number. By printing spaces, tab to that position on the current line based on the $XCOORDINATE reserved word. Note that characters typed as input to the computer are not tallied in maintaining $XCOORDINATE.

For Example:

Example	Meaning
W #,?80-$L(X)/2,X	Output a page-feed combination, and write the value of "X" centered on an 80-column page.

W 65000:2R5.2	Outputs "~~~~650.00" (where "~" represents a space character). (The first two spaces are from "2R", the field width is 5+2+1=8, the data to be printed (650.00) is six characters, leaving two more spaces to be printed before the value).
W 65000:D10	Outputs "02/16/78~~".
W 65000:2T4	Outputs "~~6500" because the value is truncated to fit the field.
W *7,"<ERROR>":3	Outputs a BELL character (see 16, Appendix I - ASCII Codes), and the message "~~~<ERROR>".

7.2 ASSIGN and UNASSIGN Commands

MIIS is designed so that applications programs can communicate to peripheral devices without regard for the physical characteristics of the devices. The ASSIGN command is used to establish ownership of a particular device, as well as to indicate to MIIS to which device it should direct its input/output. The ASSIGN command also enables the applications program to control certain physical characteristics of a particular device.

The UNASSIGN command is used to relinquish ownership of devices.

Current I/O Device

Unless otherwise specified in the course of a job's execution, the device from which input is accepted and to which output is directed is the job's PRINCIPAL DEVICE, which is the terminal from which the job was signed-on.

If an input/output command (PRINT, LOAD, READ, WRITE) is executed in direct-mode (see section 2.4, Interpreter and MIIS as a Control Language), the I/O is directed to the job's principal device. An exception: if the programmer enters a direct mode line containing an ASSIGN followed by I/O commands, MIIS directs the I/O to the ASSIGNed device. When the entire direct mode line has been executed, I/O reverts to the principal device. In addition, if the direct mode line CALLs or DOes another program, the current device established by the ASSIGN in the direct mode line will extend to the subordinate procedure.

If an I/O command is executed in a program line, the I/O is directed to the device last assigned in a program line. In other words, a previous direct-mode ASSIGN does not alter the current device for a subsequent program-line I/O command. If the current I/O device is not the principal device, and the I/O device is then UNASSIGNed, the current I/O device will revert to the principal device. If the current I/O device is the principal device and is then UNASSIGNed, subsequent output will be lost (that is, directed to thin

air), but attempts to input will result in a hung job.

The reserved word $IO contains the MIIS device number of the current I/O device. The pseudo-device "0" always refers to the job's principal device (so that ASSIGN 0 will make the principal device the I/O device).

ASSIGN Command

The ASSIGN command may be used to establish ownership of particular devices. In this case, the format is:

ASSIGN arglist

where each argument in the list is a MIIS expression that evaluates to a legal device number. Attempts to ASSIGN an illegal device number yield an error. Attempts to ASSIGN a device already assigned to another job will cause the "assigning" job to be placed on the resource queue and wait until the device is UNASSIGNed.

Certain device numbers have special meanings, as follows:

DEV	Meaning
0	Always refers to the job's principal device.
1-8	These devices correspond to various blocked devices which are present in the particular hardware configuration. Device #1 is typically a magtape or diskette unit, but any device 1-8 may be a blocked line-printer, magtape-on-disk, one of several magtapes, etc., depending on the configuration. Any device 1-8 which does not correspond to hardware may be used as a pseudo-device whose ownership can be used as a vehicle for inter-job communication.
9	Always refers to the system's view buffer, which is a disk-block image buffer that is used both as a software-locking device and as a workspace for inspecting and editing disk blocks.
10	Always refers to the system's console terminal.
11+	These devices are user terminal devices. If the system is configured with character printers, these printers are assigned device numbers sequentially beginning with #11.

Certain physical characteristics of the various system devices may be controlled by specifying arguments to the ASSIGN command in a special format:

ASSIGN dev:<parameter>=<value>:...:<parameter>=<value>

Each parameter corresponds to a different characteristic of the interaction between MIIS and the peripheral. The set of characteristics which may be controlled depends on whether the device is blocked (like magtapes, diskettes, magtape-on-disk, blocked line printers, etc.) or unblocked (like character printers, CRTs, etc.).

The following is a list of the various parameters and legal values that are employed for the two types of devices. Note that each parameter is identified by the first letter of its name when used as an argument to the ASSIGN command, and that the range of legal values is dependent on each parameter.

Unblocked Devices

Parameter Name	Value	Meaning
(A)lpha Lock	TRUE	Interpret all lower-case input as upper-case.
	FALSE	Interpret all lower-case input as lower-case.
(B)reak Inhibit	TRUE	On receipt of keyboard break, continue processing.
	FALSE	On receipt of keyboard break, issue ?BREAK error and abort processing.
(C)ontrol Characters	0	Ignore input of control characters and do not echo them.
	1	Accept input of control characters and do not echo them.
	2	Ignore input of control characters and do echo them.
	3	Accept input of control characters and do echo them.
(D)evice Type	integer	Identifies a device with an entry in the FF/CR/LF table which is maintained by the utility program "CF". MIIS is supplied with a number of these device types already defined, as referenced by the utility program/global %DV.

(E)nter Key	integer > 0	ASCII code of character which, when received as input during a READ command or direct-mode input, will terminate the input. Usually set to 13 (Carriage Return).
	0	Any control character acts as an enter key. Additionally, the enter key character is left in the input buffer unless overwritten with output characters until it is read with a single-character READ (this is to enable the applications program to determine which control key was used to enter the data). If $A(0,R)'=0, the rubout key does not act as an enter key. If $A(0,R)=0, the rubout key does act as an enter key. The line-delete key ($A(0,L)) takes precedence over the enter key.
(F)ull Duplex	TRUE	Splits the terminal buffer in half, using one half for input data, and the other for output. Enables input to be accepted from the terminal at the same time the terminal is active doing output. Data entered when the terminal is output-active will not be echoed until the terminal is no longer output-active.
	FALSE	Entire terminal buffer is used for input or for output, and any data entered when the terminal is output-active is overwritten in the buffer, and therefore lost.

(I)nter-machine Communication	TRUE	The following sequences of ASCII codes are output to the device before entering the various input-hung states: READ 13, 3 READ * 13, 5 HALT or error 13, 4, "GOOD BYE", 13, 10 Timed READ 13, 6, X, Y Timed READ * 13, 7, X, Y where "X" and "Y" above are the low-order 8 bits and the high-order 8 bits, respectively, of the binary representation of the number of seconds (so that Y*256+X is the number of seconds in the timed READ). MIIS allows an additional two seconds to satisfy the timed READ if $A(0,I)=TRUE. $A(0,D) should be set to zero for any device where $A(0,I)=TRUE.
	FALSE	No characters are output to the device before entering input-hung states.
(L)ine Delete Character	integer	ASCII code of character which, when received as input during direct-mode input or a READ command, will cause a <"?", CR/LF> to be echoed and all input so far to be deleted. If the value for this parameter is zero (0), no line-delete character exists for that device.
(N)on-Echo	TRUE	Do not echo characters which are entered from the device.
	FALSE	Echo characters which are entered from the device.
(P)rinter Flag	TRUE/ FALSE	No action - used by convention in applications programs.

(R)ubout Echo Character	integer > 0	ASCII code of character which will be echoed when a RUBOUT character is input during direct-mode input or a READ command. Each striking of the RUBOUT key will cause the most-recently entered character to be deleted.
	0	If the value of this parameter is zero (0), striking the RUBOUT key will have no effect (i.e., the data string <rubout> will be included in the input data).
(T)ab Character Handling	TRUE	Output TAB characters as the ASCII code 9.
	FALSE	Output TAB characters as a number of spaces, tabbing to eight-character stops based on the $XCOORDINATE reserved word.
(X)on-Xoff	0	Accept Xoff (CTRL/S) to stop printing, and Xon (CTRL/Q) to resume printing. Do not generate Xon-Xoff on input.
	1	Do not accept Xoff to stop printing and do not generate Xon-Xoff on input.
	2	Accept Xoff to stop printing, and Xon to resume printing. When accepting input, generate Xoff when the input buffer is almost full, and Xon when the buffer is again empty.
	3	Do not accept Xoff to stop printing. When accepting input, generate Xoff when the input buffer is almost full, and Xon when the buffer is again empty.

Blocked Devices

Parameter Name	Value	Meaning
(A)ssignment Type	integer	Identifies the blocked device with an entry in the FF/CR/LF table which is maintained by the utility program "CF". MIIS is supplied with a number of these device types already set up: for details, call MEDITECH. FF/CR/LF instructions are interpreted exactly as if they were instructions to a character device identified by $A(0,D), except that fill characters are spaces rather than rubouts. If the (F)ormatted parameter (see below) is FALSE, the (A)ssignment parameter has no meaning.
(B)reak Inhibit	TRUE	When an error is encountered in communicating with a magtape, do not issue an ?IOdev error. Instead, continue processing and leave error checking to the applications program. If the blocked device is a diskette, issue an error regardless of the value of this parameter.
	FALSE	When an error is encountered in communicating with the device, issue an ?IOdev error and abort the process.

(C)ommand

0	Instruction to a magtape or diskette to position the buffer pointer to the top of the buffer.
1	Output the contents of the in-core buffer to the device (magtape or diskette).
2	Output an <EOF> marker to the magtape.
4	Rewind the magtape. For magtape-on-disk or diskette, causes a logical rewind, repositioning the sequential file pointer to the beginning of the file.
8	Force a read from the device into the buffer.
16	Space the physical device one block forward (does not affect contents of buffer).
32	Space the physical device one block backward (does not affect contents of buffer).
64	Search the tape for the next <EOF> marker.
128	For diskettes, instructs MIIS to ascertain certain physical attributes of the diskette unit so that MIIS may communicate with it accurately. Should be the first instruction to the diskette when initializing it.

These values may be added, with the result that each value in the sum will have its usual effect. The commands are performed in the order given above (low-high). Note that "0" is an implicit part of every such sum.

(D)elimiter Character

integer	ASCII code of character which is to be used as record-separator for stream-mode (variable length records) data recording.

(E)bcdic

TRUE	Encode the data on the magtape or diskette using the EBCDIC data-encoding scheme.
FALSE	Encode the data on the magtape or diskette using the ASCII data-encoding scheme.

(F)ormatted	TRUE	Allow programs to use the "colon" syntax in the WRITE command (see section 7.1, READ and WRITE Commands) to justify fields on the magtape or diskette. Also allows tabbing with the ?tab operator, and enables reference to the (A)ssignment type parameter outlined above.
	FALSE	All forms-control operators (WRITE #, WRITE !, and WRITE ?tab) and attempts to use the "colon" syntax in the WRITE command are interpreted as no-ops.
(H)igh Density	TRUE	For magtape devices whose density is software-selectable, use the higher of the two available densities (typically 1600 BPI).
	FALSE	For magtape devices whose density is software-selectable, use the lower of the two available densities (typically 800 BPI).
(L)ogical Record Size	0	Record information on the tape or diskette using variable record size. Each argument to a WRITE command (and each program line output by a PRINT command) constitutes an individual record on the tape. These records are separated by the delimiter specified by the (D)elimiter parameter. Records output in this way may span two or more physical tape blocks. This mode is called "stream mode".
	integer > 0	Records are of fixed length, and no delimiter character is inserted between records. The logical record size must evenly divide the physical record size, as logical records may not span physical blocks in this mode.
(P)hysical Record Size	integer	Specifies how much of the tape buffer is to be used, and corresponds to the block size used on the tape. This number is arbitrary, but must be an even multiple of the logical record size described above. $A(1,P) should be an even number. When reading from or writing to the tape, attempts to move the pointer beyond the end of the buffer as specified by this parameter will cause the next block on the tape to be read into or written from the buffer.

(R)ead Maximum	integer < 255	The maximum number of characters to be read from the tape or tape buffer on a READ command. Used to enable processing of tapes with logical record sizes greater than 255 characters (since MIIS cannot handle (name,data) pairs with combined length greater than 255 characters).
(T)ab Character	TRUE	Output TAB characters as the ASCII code 9.
	FALSE	Output TAB characters as a number of spaces, tabbing to eight-character stops based on the $XCOORDINATE reserved word.

The following are examples of how these parameters are used:

Example	Meaning
A 0:N=1:F="":B=1	Make a principal device non-echo, not full-duplex, and unbreakable.
A 0:E=10:L=27	Make the enter key NEWLINE and the line-delete key ESCAPE.
A 1:P=1024:L=512:R=128	For a magtape device, make the buffer size 1024 bytes, the logical record size 512 bytes, and the read maximum 128 bytes (so that it takes four READs to input a single logical record).

UNASSIGN Command

The UNASSIGN command is used to relinquish ownership of a device. It should not be assumed that any of the device's characteristics as established by the ASSIGN command will be retained once the device is unassigned.

The UNASSIGN command takes a list of arguments:

UNASSIGN arglist

where each argument in the list is an expression that evaluates to a legal device number. Attempts to UNASSIGN a device that is not owned by the job result in no action. Attempts to explicitly UNASSIGN device #0 give an error (the principal device must be UNASSIGNed directly by its number).

When an UNASSIGN command is issued, all jobs on the resource queue are placed on the CPU wait queue to try again to own the resource for which they are contending.

7.3 $ASSIGNMENT and $JOB

$ASSIGNMENT Reserved Word and Function

The $ASSIGNMENT function is used to test the current values of the various assignment parameters outlined in section 7.2, ASSIGN and UNASSIGN Commands.

The format of the function is:

$ASSIGNMENT(device,parameter-name)

and the function returns either <rubout>, <nil>, or an <integer>, depending on the range of values.

The $ASSIGNMENT reserved word returns $A(0,D)#4, plus four if $A(0,B) is TRUE, plus eight if $A(0,N) is TRUE.

For Example:

Example	Meaning
$A(1,P)	Returns an integer describing the buffer size for device #1.
$A(23,A)	Returns <nil> or <rubout>, depending on whether device #23 is locked on upper-case letters or not.

$JOB Function and Reserved Word

The $JOB reserved word returns as its value the job number of the job that is evaluating the reserved word.

The $JOB function takes a single argument:

$JOB(device)

and returns the job number of the job that owns the device. If the device is not owned, the $JOB function returns zero (0).

For example:

```
A:$J(DEV)=0 DEV I $J(DEV)'=$J W *7,"<DEVICE IN USE>":3
```

is the correct way to interactively gain control of a particular device.

7.4 View Buffer, TRANSFER Command, $WORD and $BYTE

View Buffer

The view buffer is a 512-byte buffer contained in low memory. The location of the view buffer is listed in the in-core table at $W($K+4) (i.e., the value of $W($K+4) is the first word of the view buffer).

MIIS device number nine (9) is always identified with the view buffer. While the view buffer is not properly an I/O device, having the buffer assigned ensures that access to the buffer will be limited to one job at a time.

The view buffer serves two purposes:

1) Acts as a software lock. The following MIIS commands require that the issuing job own the view buffer:

   SET $WORD and SET $BYTE
   TRANSFER IN (T <) and TRANSFER OUT (T >)

   Note that SET $WORD, SET $BYTE and TRANSFER OUT also require that the system WRITE switch be set.

2) Acts as a disk-block image buffer. Individual disk blocks may be copied into the view buffer, examined, altered, and copied back out to the disk.

TRANSFER Command

The TRANSFER command is used to copy blocks from the disk to the view buffer, and from the view buffer to the disk.

The TRANSFER command takes an argument list. Each argument in the list consists of one of the following:

```
<EXP:EXP
>EXP:EXP
<EXP
>EXP
```

The carets ("<" and ">") indicate the direction of the transfer. The less-than sign "<" means TRANSFER INto the view buffer from the disk. The greater-than sign ">" means TRANSFER OUT from the view buffer to the disk.

As seen above, the block number on the disk is specified in one of two ways: EXP (which evaluates to an absolute disk block number) or EXP:EXP (which evaluates to a disk drive number (0-7) and a relative disk block number on that drive (drive:block)).

Attempts to use the TRANSFER command without proper software locks (view buffer assigned and WRITE switch set) will give a ?SYSTM error. Attempts to access an non-existent disk block will yield an ?IODSK error.

For example:

Example	Meaning
T <1:4096	Copy the 4096th block on drive #1 into the view buffer.
T >654321	Copy the contents of the view buffer out to the disk, to the 654321st block of the system (regardless of the physical drive on which that block is stored).

$WORD and $BYTE Functions

The $WORD and $BYTE functions are used to inspect and alter specific memory locations. Both functions take a single argument:

$WORD(nexp) and $BYTE(nexp)

The expression "nexp" should evaluate to a number corresponding to an existing memory location. Locations for $WORD are word locations; locations for $BYTE are byte locations. These locations are numbered sequentially starting from location zero.

$WORD and $BYTE may appear on the left-hand-side of a SET command, in which case, the memory location is set to the value specified on the right-hand-side of the SET command. $WORD and $BYTE may also be used as a regular MIIS expression. For example:

Example	Meaning
SET $WORD(I)=J	Change the value of memory word-location "I" to the value of "J". Note that the view buffer must be assigned and the system WRITE switch must be set in order for the command to be successfully executed.
IF $BYTE(I)=J	Compare the value stored in memory byte-location "I" with the value of "J".

IMPORTANT: The argument (memory locations) and value (contents of the memory locations) are both specified as DECIMAL (not octal or hexidecimal) integers.

CHAPTER 8: PROGRAM FLOW AND LOGIC

8.1 GOTO and OVERLAY Commands

GOTO Command

The GOTO command is used to transfer control from one part of a MIIS program to another part of the same program. Control is transferred in a one-way manner, without stacking a return address as in a subroutine.

Control may be transferred from any point in a program to the beginning of any program line in the same program. The syntax is as follows:

GOTO arglist

where each argument in the list is one of the following:

Argument	Meaning
TAG	Transfer control to the line labelled "TAG" (see section 4, Programs - Input, Edit, Delete for a discussion of TAGs).
TAG:EXP	Evaluate EXP. If TRUE, make the indicated transfer. If FALSE, do not make the indicated transfer.

Attempts to GOTO an undefined tag yield an ?UNDEF error.

For example:

Example	Meaning
G TAG	Transfer control to the line labelled "TAG".
G TAG:A,TAG1:B	If "A" evaluates TRUE, transfer to "TAG". If "B" evaluates TRUE, transfer to "TAG1". If both are false, make no transfer.
G:A TAG:B,TAG1:C	If "A" evaluates FALSE, make no transfer. If "A" is TRUE and "B" is TRUE, transfer to "TAG". If "A" is TRUE and "C" is TRUE, transfer to "TAG1".

OVERLAY Command

The OVERLAY command is used to transfer control from one part of a MIIS program to another part of a different program. Control is transferred in a one-way manner, without stacking a return address as in a subroutine.

Control may be transferred from any point in a program to the beginning of any program line in another program. The syntax is as follows:

OVERLAY arglist

where each argument in the list is one of the following:

Argument	Meaning
PGM	Transfer control to the first line of the specified program.
PGM@TAG	Transfer control to the line labelled "TAG" in program "PGM".
PGM:EXP or PGM@TAG:EXP	Evaluate EXP. If TRUE, make the indicated transfer. If FALSE, do not make the indicated transfer.

Attempts to OVERLAY to an undefined program or tag yield an ?UNDEF error. Attempts to execute an OVERLAY command while within the scope of a DO statement will result in a ?CMMND error.

For example:

Example	Meaning
O PGM@TAG	Transfer control to program "PGM", line "TAG".
O PGM:A,PGM1:B	If "A" evaluates TRUE, transfer to "PGM". If "B" evaluates TRUE, transfer to "PGM1". If both are false, make no transfer.
O:A PGM@TAG:B,PGM@TAG1:C	If "A" evaluates FALSE, make no transfer. If "A" is TRUE and "B" is TRUE, transfer to "PGM@TAG". If "A" is TRUE and "C" is TRUE, transfer to "PGM@TAG1".

8.2 DO, CALL and QUIT Commands

DO Command

The DO command is used to transfer control from one part of a MIIS program to another part of the same program. Control is transferred in such a way that a matching QUIT command will return control to the MIIS command immediately following the original DO command. In this way, applications can create and use subroutines.

There is a capability to pass arguments into a subroutine, and to pass to a subroutine the names of variables in which to return results. The mechanism for this is is explained in detail in section 8.3, Subroutines.

Control may be transferred from any point in a program to the beginning of any program line in the same program. The syntax is as follows:

DO arglist

where each argument in the list is one of the following:

Argument	Meaning
TAG	Transfer control to the line labelled TAG.
TAG:EXP	Evaluate EXP. If TRUE, make the indicated transfer. If FALSE, do not make the indicated transfer.
TAG[arglist] or TAG[arglist]:EXP	Make the transfer (conditionally on EXP) and create local variables %0-%n that correspond to the values being passed. This process is further explained in section 8.3, Subroutines.

For example:

Example	Meaning
D TAG	Transfer control to the line labelled "TAG". When a matching QUIT command is encountered, return control to the instruction immediately following the DO.
D TAG:A,TAG1:B	If A evaluates TRUE, transfer to subroutine TAG. After returning, if B evaluates TRUE, transfer to subroutine TAG1.

D:A TAG:B,TAG1:C	If A evaluates FALSE, make no transfer. If A evaluates TRUE and B evaluates TRUE, transfer to subroutine TAG; after returning, if C evaluates TRUE, transfer to subroutine TAG1.

Attempts to DO an undefined tag yield an ?UNDEF error.

Note that the IF-SWITCH (see section 8.4, IF, ELSE and IF-SWITCH) is stacked along with the subroutine call, so that IF commands and TIMED READs that occur within the subroutine do not affect the value of the IF-SWITCH in the parent routine after control returns.

CALL Command

The CALL command is used to transfer control from one part of a MIIS program to another part of a different program. Control is transferred in such a way that a matching QUIT command will return control to the MIIS command immediately following the original CALL command in the original program. In this way, applications can create and use subroutines.

There is a capability to pass arguments into a subroutine, and to pass to a subroutine the names of variables in which to return results. The mechanism for this is is explained in detail in section 8.3, Subroutines.

Control may be transferred from any point in a program to the beginning of of any program line in another program. The syntax is as follows:

CALL arglist

where each argument in the list is one of the following:

Argument	Meaning
PGM	Transfer control to the first line of the specified program.
PGM@TAG	Transfer control to the line labelled TAG in program PGM.
PGM:EXP or PGM@TAG:EXP	Evaluate EXP. If TRUE, make the indicated transfer. If FALSE, do not make the indicated transfer.
PGM[arglist] or PGM[arglist]:EXP or PGM@TAG[arglist] or PGM@TAG[arglist]:EXP	Make the transfer (conditionally on EXP) and create local variables %0-%n that correspond to the values being passed. This process is further explained in section 8.3, Subroutines.

For example:

Example	Meaning
C PGM	Transfer control to program PGM. When a matching QUIT command is encountered, return control to the instruction immediately following the CALL in the CALLing program.
C PGM:A,PGM1:B	If A evaluates TRUE, transfer to subroutine PGM. After returning, if B evaluates TRUE, transfer to subroutine PGM1.
C:A PGM:B,PGM1:C	If A evaluates FALSE, make no transfer. If A evaluates TRUE and B evaluates TRUE, transfer to subroutine PGM; after returning, if C evaluates TRUE, transfer to subroutine PGM1.

Attempts to CALL an undefined program or tag yield an ?UNDEF error.

A WARNING: When MIIS executes a CALL command, it retains in an internal stack the name of the program in which the CALL was made, and a byte offset indicating the distance between the CALL command and the beginning of the program. When the corresponding QUIT is executed, this information is retrieved from the stack. The appropriate program is located by name, either by finding it in a program buffer or retrieving it from the disk; and the appropriate number of characters in the program are skipped in order to resume processing with the instruction immediately following the CALL command. If the CALLing program is changed and re-FILEd between the time the CALL is executed and the QUIT is encountered, the byte offset may no longer be valid, with unpredictable results. (The most usual result is a ?SYNTX error). If the CALLing program is not FILEd before the CALL is made, the return from the CALL will again give a ?SYNTX or ?UNDEF error.

Note that the IF-SWITCH (see section 8.4, IF, ELSE and IF-SWITCH) is stacked along with the subroutine call, so that IF commands and TIMED READs that occur within the subroutine do not affect the value of the IF-SWITCH in the parent routine after control returns.

QUIT Command

The QUIT command is used to terminate the action of a DO, CALL or FOR command. When encountered within the scope of a FOR command-line, the QUIT command causes the FOR to be terminated. When encountered within the scope of a DO or CALL, the DO/CALL is terminated, any %-variables are deleted, and control is returned to the parent (calling) routine.

The QUIT command takes no arguments:

```
QUIT
```

and like all commands except IF and FOR, the QUIT may be made conditional as follows:

```
QUIT:EXP
```

For example:

```
FOR I=1:1:10 FOR J=1:1:10 QUIT:J>I
```

will loop 1+2+3+4+5+6+7+8+9+10=55 times, because when the QUIT is executed the inner FOR statement will terminate back to the outer one.

Also,

```
        DO A,B QUIT
A       WRITE "A" QUIT
B       WRITE "B" QUIT
```

will output "AB", and then either:

1) Return to direct-mode ("?" prompt) if the above code was a stand-alone program invoked in direct-mode.

2) Return to a parent routine, if the above code was CALLed or DOne from another segment of MIIS code.

3) Sign off ("GOODBYE") if the above code was a stand-alone program invoked in user mode.

8.3 Subroutines

Argument Passing

In order to increase the utility of subroutines, MIIS enables the following capabilities:

1) To allow the calling (parent) program to pass values into the subroutine without regard for how the values are accessed (i.e., by what name they are called) inside the subroutine, and

2) To allow the subroutine to return values to the parent routine without regard for the names that the returned values will take in the parent routine.

This is accomplished as follows:

1) CALL and DO commands may specify a list of values that are to be passed into the subroutine. This specification takes the form of a list of MIIS expressions enclosed in square-brackets. The expressions in the list are separated by commas. There may be from 0-9 values passed in any CALL/DO:

   CALL/DO subroutine[exp,exp,exp,exp,exp,exp,exp,exp,exp]

2) When the CALL/DO is executed, the expressions in the list are evaluated, left-most expression first. After each expression is evaluated, its value is stored in a local variable called %1, %2, ..., %9 corresponding to the sequential position of the expression in the list of arguments. Each of these local variables is "stacked" as if it had been defined with the VARIABLES command. After all the expressions in the list have been evaluated, a variable called %0 is created (stacked) which has as its value the number of arguments that were passed (1-9) in this subroutine reference.

   If there is no list of arguments passed in a particular CALL/DO, no %0 variable is created.

   Note that since each %-variable is created immediately after the corresponding expression is evaluated, using %-variables in the expressions can give undesired results.

3) The %-variables may then be used in the subroutine just like any other MIIS variables. In particular, if a %-variable has as its value the name of another MIIS variable, the subroutine can use an indirect set to pass a value back to the parent routine.

4) When the QUIT that terminates the CALL/DO is encountered, MIIS first determines whether any arguments were passed on this CALL/DO. If there were, MIIS then uses the current value of %0 to determine how many %-variables need to be un-stacked (killed) before returning control to the parent routine.

For example, the following two segments of code are functionally equivalent:

```
        D SUB[A,B] Q                        V %1=A,%2=B,%0=2 D SUB Q
SUB     Q                           SUB     K %0,%1,%2 Q
```

Here are two examples of how argument passing may be used:

```
        D C["HEADER"]
        ...
C       W #,!,?80-$L(%1)/2,%1 Q ;PRINTS HEADER CENTERED ON NEW PAGE
```

also,

```
        D STRP[A,"A"],STRP[$I(A),"A"] S A=$I(A) ;STRIP LEADING, TRAILING SPACES
        ...
STRP    ;STRIPS LEADING SPACES
        V A F A=1:1 Q:$E(%1,A)'=" "
        V %9=$E(%1,A,299) K A S @%2="%9" K %9 Q
```

Note that in the last example, the variable %9 is introduced as a temporary store for the result before it is indirectly SET back into the variable A. If instead the result had been set back into A without using %9, for example:

```
S @%2="$E(%1,A,299)" K A Q  (or)  K A S @%2="$E(%1,A,299)" Q
```

the result would have been passed back incorrectly in the wrong "copy" of A. The use of %9 is a convention that circumvents this problem. The other side of the convention is of course to not use %9 as the name in which results are to be passed back from a subroutine.

Modularity of Subroutines

Passing values into and out of subroutines in only part of the requirements for effective subroutines. It is also necessary that the subroutine have a clean workspace for its intermediate results, and for that workspace not to interfere with the workspaces of the subroutine's superior or subordinate routines.

To this end, MIIS includes the VARIABLES command (see section 9.1, SET and VARIABLES Commands), which is often used as follows:

```
        DO SUB[ARG,"ARG"]
        ...
SUB     VARIABLES A,B,C
        ...
        VARIABLES %9=C KILL A,B,C SET _%2_"=%9" KILL %9 QUIT
```

Note that this technique (similar to the one used in the previous example) allows the subroutine to use the symbols A, B, and C for its calculations without regard for the meanings of those symbols in the parent routine.

8.4 IF, ELSE and IF-SWITCH

IF Command

The IF command is used to make the execution of a line (or trailing part of a line) of code conditional on the value of an expression.

The IF command may take a list of arguments or may take zero arguments:

```
    IF arglist
or, IF
```

When arguments are present in the IF command, each argument is a MIIS expression.

Both forms of the IF command act as follows:

1) If the IF is "true", MIIS then executes all commands after (to the right of) the IF command on the line on which the IF was issued. Additionally, the internal IF-SWITCH is set TRUE.

2) If the IF is "false", MIIS does not execute any commands after (to the right of) the IF command on the line on which the IF was issued. Additionally, the internal IF-SWITCH is set FALSE.

In order to ascertain whether an IF command with arguments is "true", MIIS evaluates the first expression in the list. If the expression yields a TRUE value, MIIS then evaluates the next expression in the list. If any expression yields a FALSE value, the value of the IF is FALSE, and the remaining expressions in the list are not evaluated. If all expressions yield a TRUE value, the value of the IF is TRUE.

In order to ascertain whether an IF command without arguments is "true", MIIS tests the value of the internal IF-SWITCH.

IF SWITCH

The IF-SWITCH is set by the following MIIS actions:

* Execution of an IF command
* Execution of a TIMED READ command
* Execution of a TIMED OPEN command

A new copy of the IF-SWITCH is created every time a DO or CALL command is executed. The value of the new IF-SWITCH is inherited from that in the parent routine. When the corresponding QUIT is executed, the IF-SWITCH reverts to its previous value (i.e., is unstacked).

Note that the IF-SWITCH is NOT affected by the determination of whether to execute a conditional command (like COMMAND:LEXP ARGLIST).

Therefore, until the subroutine executes its own IF command, the subroutine may access the value of the IF-SWITCH that was created by the parent routine. However, any IF commands executed by the subroutine will not interfere with the value of the parent routine's IF-SWITCH when control is returned.

ELSE Command

The ELSE command is used to test the negative sense of the IF-SWITCH.

The ELSE command takes no arguments, and may only be executed in a program line. This is to distinguish the ELSE command from the argumentless ERASE command, which may only be executed in direct mode.

If the value of the IF-SWITCH is FALSE, code to the right of the ELSE command on the line on which the ELSE command is present is executed. If the value of the IF-SWITCH is TRUE, code to the right of the ELSE command is not executed.

For example:

```
I 1 W 1                    Would output "1".
E  W 2

I "" W 1                   Would output "2".
E  W 2
```

8.5 FOR Command

The FOR command is used for simple iteration through a loop. The command takes a list of arguments:

FOR VAR=arglist

where "VAR" is the name of an index variable which takes on values as specified by the argument list:

Argument	Meaning
EXP	Assign the value of EXP to VAR.

Below, NEXPn refers to an expression that evaluates to a number:

NEXP1:NEXP2	Assign the value of NEXP1 to VAR. On subsequent iterations, increment the value of VAR by NEXP2. NEXP1 and NEXP2 are evaluated only once, so that any alterations to their values in the scope of the FOR will not affect the number of iterations.
NEXP1:NEXP2:NEXP3	Assign the value of NEXP1 to VAR. On subsequent iterations, increment the value of VAR by NEXP2. If the value of VAR exceeds NEXP3 (or, if NEXP2<0, if the value of VAR is less than NEXP3) prior to a given iteration, go on to the next argument. NEXP1, NEXP2 and NEXP3 are evaluated only once, so that any alterations to their values in the scope of the FOR will not affect the number of iterations.

The scope of the FOR statement is from the beginning of the command to the end of the command line on which it is issued: that is, any code between the argument list that is part of the FOR command and the end of the line will be executed for each loop (and for each value given to the index variable).

A GOTO or OVERLAY command executed in the scope of a FOR command will cause control to transfer accordingly, and the FOR command to be terminated.

A CALL or DO command executed in the scope of a FOR command will cause control to transfer accordingly, and will return to inside the FOR command when the corresponding QUIT command is executed.

A QUIT command executed in the scope of a FOR command will cause that FOR command to be terminated, and control passed either to the next outer-most FOR command (if there is one), or to the next line of code in the program.

FOR commands may be nested, either by being on the same line of code, or by embedding successive FOR commands inside subroutines invoked from within a FOR command.

For example:

Example	Output
`F A=1:1:9,"A",1:1:9 W A`	123456789A123456789
`F A=9:-1:1 F B=1:1 W A,B Q`	9181716151413121 11
`F A=1:1 Q:A>5 W A:1 F B=1:1 Q:B>5 W B`	112345 212345 312345 412345 512345
`F A=1:1:0,"A",0:1:1 W A`	A01
`F A=1:1:10 S A=A+1 W A:1`	2 4 6 8 10

CHAPTER 9: DATA ASSIGNMENT AND DETERMINATIONS

9.1 SET and VARIABLES Commands

SET Command

The SET Command is used to assign values to data names. Data values may be inserted in both the global data base and in a job's local symbol table. Insertions into the local symbol table are to the "current" copy of each symbol, rather than stacking a new copy. Attempts to use more than 2048 bytes of local-storage-plus-user-stack (see section 2.1, Allocation of Computer Memory) will result in a ?STORE error.

The SET command takes a list of arguments:

SET arglist

Where each argument is one of the following:

Argument	Meaning
VAR=EXP	If there exists a copy of the identifier VAR, assign the value of EXP to that identifier. If no such identifier exists, create one with the value of EXP.
VAR	Assign a <nil> value to the identifier VAR.
$UPDATE(VAR{,EXP})	(Where VAR is not a packed data field) Evaluate VAR as a number. If the optional syntax {,EXP} is present, evaluate EXP as a number, add the result to the value of VAR, and store the result in VAR. If the optional syntax {,EXP} is not present, add one (1) to the value of VAR, and store the result in VAR. (See section 9.3, $UPDATE Function).
*VAR	See section 9.5, STRUCTURED SET Commands for a discussion of this syntax.

For example:

Example	Meaning
S A	Assign the value <nil> to local variable A.
S ^A(A)DATE(DATE)=B	Set the value of B into the indicated global node.
S $U(A)	Increment A by one and store the new value in A.

VARIABLES Command

The VARIABLES command is used to assign values to data names in a job's local symbol table. Insertions via the VARIABLES command are always to a newly-created copy of the identifier specified in the argument. This newly created copy is created by pushing down the old copy so that the "current" value of that identifier is accessed by references to that identifier. A subsequent KILL of that identifier will restore the old value of the identifier. Pushed-down copies of local symbols continue to occupy space in the local symbol table. Attempts to use more than 2048 bytes of local-storage-plus-user-stack (see section 2.1, Allocation of Computer Memory) will yield a ?STORE error.

The VARIABLES command takes a list of arguments:

VARIABLES arglist

where each argument in the list is one of the following:

Argument	Meaning
VAR=EXP	(Where VAR is a local symbol and is not a packed-data field) Create a new copy of identifier VAR on top of any existing copies, and assign the value of EXP to that variable.
VAR	(Where VAR is a local symbol and is not a packed-data field) Create a new copy of identifier VAR on top of any existing copies, and assign the <nil> value to that variable.

For example, consider the following terminal session:

```
?S A=1 W A
1
?V A=2 W A
2
?K A W A
1
?
```

Use of the VARIABLES command is indicated in the following circumstances:

1) Applications which require a push-down stack of values which may be popped in a LIFO manner. Many recursive routines fit this description.

2) Subroutines which will use local storage for workspace often use the VARIABLES command to ensure that the subroutine's workspace will not interfere with that of the parent routine (see section 8.3, Subroutines).

9.2 KILL Command

The KILL command is used to delete identifiers and their values from local and global storage.

The KILL command takes either zero arguments or a list of arguments:

```
    KILL
or, KILL arglist
```

The effect of a KILL command with zero arguments is to delete all identifiers and their values from the local symbol table. This includes any defined %-variables, any defined NAMESPACE names, and the $G reserved word.

If arguments are present in the KILL command, each argument has one of the following forms:

Argument	Meaning
VAR	Delete the identifier VAR and its value from the global data base or delete the current copy of VAR from the local symbol table. If VAR is a packed-data field, delete the field and its delimiter from the string, and decrement the zero'th field by one (see section 6.3, Data Names).

*VAR	"Structured KILL". If VAR is a global variable, delete the identifier VAR and its value, as well as any data structures subordinate to the identifier VAR (see section 6.3, Data Names). If VAR is a local variable, delete the current copy of VAR and the current copies of any data structures subordinate to VAR.
(VAR{,VAR})	"Exclusive KILL". Delete all local symbols except: * Any %-Variables, * NAMESPACE names 1-9 and $G reserved word, * All copies of the variables named inside the parentheses.
(*VAR{,*VAR})	"Structured Exclusive KILL". Delete all local symbols except %-variables, NAMESPACE names, the $G reserved word, the identifiers named inside the parentheses, and structures subordinate to those identifiers.

For example, after the following:

```
K  S A=2,A(1)=2,B=2 V A=1,A(1)=1,B=1
```

then the results of the following KILL commands are as follows:

Example	Remaining Variables
K A	A=2, A(1)=1, A(1)=2, B=1, B=2
K *A	A=2, A(1)=2, B=1, B=2
K (A)	A=1, A=2
K (*A)	A=1, A=2, A(1)=1, A(1)=2

9.3 $UPDATE Function

The $UPDATE function is used to increment or decrement numeric-valued variables.

The $UPDATE function takes one or two arguments:

$UPDATE(VAR{,EXP})

The first argument is any legal MIIS identifier EXCEPT a packed-data field. The second argument, if present, is an expression that evaluates to a number, and represents the value to be added to the variable specified by the first argument. If the second argument is not present, the added value defaults to one (1).

Evaluation of the $UPDATE function causes the following sequence of events:

1) The variable VAR is evaluated.

2) If the second argument EXP is present, it is evaluated.

3) The value of EXP (if present) or one (if EXP is not present) is added to the value of VAR.

4) The result of step (3) is stored in variable VAR, and is returned as the value of the function.

The $UPDATE function offers the following advantages over more traditional methods of incrementing and decrementing variables:

1) The entire evaluation of the function takes place in a time-sharing-protected fashion, so that it is not necessary to use OPENs when $UPDATEing global nodes that might be subject to time-sharing conflict.

2) The $UPDATE function makes only a single reference to the identifier, and is therefore faster than a fetch-add-store sequence.

3) The $UPDATE function often requires fewer characters of code than the traditional alternatives.

Some examples of using $UPDATE are as follows:

```
A       I $U(CNT)<END ... G A

        (or)

        S ^A($U(^A))=DATA

        (or)

        S $U(^COUNT(DATE,HOUR),NUM)
```

9.4 NAMESPACE Command

NAMESPACEs are used to abbreviate references to local and global variables, by allowing part of the name of the variable to be replaced by a NAMESPACE identifier. These identifiers are stored in the local symbol table as if they were local variables. There are a maximum of nine such identifiers, each represented by a digit 1-9. The identifiers are defined and undefined (deleted) by the NAMESPACE command.

The NAMESPACE command takes either zero arguments or a list of arguments:

```
    NAMESPACE
or, NAMESPACE arglist
```

If given with zero arguments, the effect of the NAMESPACE command is to delete all namespace identifiers from the symbol table.

If given with a list of arguments, each argument takes one of the following forms (where 'n' below represents a digit 1-9):

Argument	Meaning
n	Delete NAMESPACE #n.
n=VAR	Establish NAMESPACE #n as the name VAR, where VAR is a legal MIIS identifier and not a packed-data name.
VAR	Establish NAMESPACE #1 as the name VAR, where VAR is a legal MIIS identifier and not a packed-data name.

Using Namespaces

Once established, NAMESPACEs may be used as abbreviations for the leading part of an identifier by replacing the abbreviated part with "@n", where 'n' is the namespace number.

For example, after executing:

```
N 1=X,2=X(1)
```

there exist the following sets of equivalent references:

```
@1 = @ = X
@1(1) = @2 = X(1)
@1X = @X = X("X")
@1(1)X(1) = @2X(1) = X(1)X(1) = X(1,"X",1)
```

9.5 STRUCTURED SET Commands

The STRUCTURED SET commands enable the copying of data structures from one identifier to another identifier. In addition, they allow the programmer to create new data structures that represent the intersection, union, or set-subtraction of two existing data structures.

All global arrays created by STRUCTURED SET commands are optimally efficient - that is, the number of actually-used bytes per level 0 disk block is maximized. This is discussed further in section 13.2, Physical Storage of Globals and Programs.

STRUCTURED SET commands are a subset of the available SET command syntax, and are actually just another type of argument to the SET command (see section 9.1, SET and VARIABLES Commands):

```
SET arglist
```

where each argument is one of the following (VAR below refers to any legal identifier name except a packed-data field):

Argument	Meaning
*VAR	(Where VAR is a global identifier). Copy the entire structure subordinate to VAR (including VAR itself) back into VAR in such a way that the new physical representation of the data is optimally efficient in terms of storage used.
*VAR1=VAR2	Copy the entire structure subordinate to VAR2 (including VAR2 itself) into VAR1, merging the new structure with any structure already subordinate to VAR1.

*VAR1=VAR2&VAR3	Intersect the structures subordinate to VAR2 and VAR3, and create a structure under VAR1 with nodes defined where VAR2 and VAR3 intersect. The values of the newly defined nodes are <nil>. Count the number of new nodes created, and store that number in VAR1.
*VAR1=VAR2!VAR3	Union the structures subordinate to VAR2 and VAR3, and create a structure under VAR1 with nodes defined representing that union operation. The values of the newly defined nodes are <nil>. Count the number of new nodes created, and store that number in VAR1.
*VAR1=VAR2'VAR3	Set-subtract the structure subordinate to VAR3 from the structure subordinate to VAR2, and create a structure under VAR1 with nodes defined representing that set-subtraction operation. The values of the newly defined nodes are <nil>. Count the number of new nodes created, and store that number in VAR1.

For example, if we start with:

```
^A(1), ^A(2), ^A(2,1), ^A(3), ^A(4)
^B(2), ^B(4), ^B(6)
```

then:

Example	Result
S *^A	Compact ^A to use storage optimally.
S *C(1)=^A	Create C(1,1); C(1,2); C(1,2,1); C(1,3); C(1,4) all equal to the <nil> value.
S *C=^A&^B	Create C(2) and C(4) equal to the <nil> value. Since two nodes were created, set C=2.
S *C=^A!^B	Create C(1), C(2), C(2,1), C(3), C(4), and C(6) all equal to the <nil> value. Since six nodes were created, set C=6.
S *C=^A'^B	Create C(1), C(2,1), C(3) all equal to the <nil> value. Since three nodes were created, set C=3.

9.6 $DEFINE and $NEXT Functions

$DEFINE Function

Since a reference to an undefined variable returns a <nil> value, it is not possible to determine whether an identifier is defined with a <nil> value or not defined at all. The $DEFINE function exists to solve this problem.

The $DEFINE function takes one or two arguments:

$DEFINE(VAR1{,VAR2})

VAR1 is the name of the identifier that is being tested for existence, and may be any identifier except a packed-data name. VAR2, if present, is any identifier except a packed-data name and is the name of the variable in which to store the value assigned to VAR1.

The $DEFINE function returns TRUE if VAR1 is defined, and FALSE if VAR1 is not defined.

If VAR1 is not defined and VAR2 is present, the <nil> value is assigned to VAR2.

For example, assume A="", B=1, C=2. Then:

Example	Result
$D(A,B)	Returns TRUE, and B="".
$D(C,A)	Returns TRUE, and A=2.
$D(D)	Returns FALSE.
$D(D,B)	Returns FALSE and B="".

$NEXT Function

In section 6.3, Data Names, it was mentioned that subscripted arrays can be used to represent tree-structured databases. The analogy works because each subscript may be viewed as one of potentially many branches of the tree that descend from a particular node. Each node is uniquely identified by the subscripts that precede it in the identifier name.

The $NEXT function is used to step through the branches that descend from a particular node in the tree. This corresponds to stepping through all defined subscripts at a particular subscript position in the identifier. These subscript positions are called subscript "levels".

For example, if A(1,2), A(1,3), A(1,4), A(5,6), and A(5,7) are defined, stepping through the subscripts at the first level would identify a subscript valued 1 and a subscript valued 5. If we choose the first of these, we may step through the subscripts descending from the node A(1). There are three of these: 2, 3, and 4. If instead we choose the second first-level subscript (5) and step through the subscripts descending from the node A(5), there are two: 6 and 7.

The $NEXT function takes one or two arguments:

$NEXT(VAR1{,VAR2})

If VAR1 is a global scalar variable, the name of the next (in collation sequence order: see section 6.4, Collation of Data Names) defined global scalar variable is returned. If no such variable exists, the <nil> value is returned. If VAR2 is present, the value of the "next" variable is assigned to VAR2.

If VAR1 is a local scalar variable, the name of the next defined local scalar variable is returned. If no such variable exists, the <nil> value is returned. "Pushed-down" copies of local variables are ignored. If VAR2 is present, the value of the "next" variable is assigned to VAR2.

If VAR1 is a global subscripted variable, the function returns the name of the next (in collation sequence order) defined subscript at the same level as the right-most subscript of VAR1. If no such subscript is defined, the function returns the <nil> value. If VAR2 is present, the value stored at that subscript position is assigned to VAR2. If there is no such value or no such subscript, <nil> is assigned to VAR2.

If VAR1 is is a local subscripted variable, the function returns the name of the next defined subscript at the same level as the right-most subscript of VAR1. Copies of local variables "pushed-down" by the VARIABLES command are ignored. If no such subscript is defined, the function returns the <nil> value. If VAR2 is present, the value stored at that subscript position is assigned to VAR2. If there is no such value, or no such subscript, <nil> is assigned to VAR2.

For example, if the following local and global data is defined:

```
B="Z"                   ^B="Z"
B(1,2)="Y"              ^B(1,2)="Y"
B(1,2,3)="X"            ^B(1,2,3)="X"
B(1,2,3)="W"
B(1,2,4)="V"            ^B(1,2,4)="V"
```

then, the following $NEXTs will return:

Example	Result
$N(A,C)	Returns "B", and sets C="Z".
$N(^A,C)	Returns "^B", and sets C="Z".
$N(B(""),C) or $N(^B(""),C)	Returns 1, and sets C="" (since no value is stored at B(1) or ^B(1)).
$N(B(1,"")) or $N(^B(1,""))	Returns 2.
$N(B(1,2,3),C) or $N(^B(1,2,3),C)	Returns 4, and sets C="V".
$N(B(1,2,4),C) or $N(^B(1,2,4),C)	Returns <nil>, and sets C="".

9.7 $GLOBALGET and $G Reserved Word

While subscripted arrays may be used to represent tree-structured databases (as discussed in sections 9.6, $DEFINE and $NEXT Functions and 6.3, Data Names), it is sometimes true that a file is best treated in a sequential or serial manner.

The $GLOBALGET function permits sequential access to a local or global data structure. This access method is essentially a "physical" $NEXT, which returns the next (in collation sequence order) defined data name/value pair regardless of the number of subscripts in the next data name.

$G Reserved Word

The $G reserved word is an auxilliary to the $GLOBALGET function and is a set of packed-data fields such that:

$G[0]	= Number of fields in the $G reserved word.
$G[1-thru-($G[0]-1)]	= Running names of subscripts, changed by the $GLOBALGET function each time it is evaluated.
$G[$G[0]]	= The data value stored at the name VAR($G[1],...,$G[$G[0]-1]).

Fields of the $G reserved word may be SET or KILLed like any packed-data name. The $G reserved word is stored in the local symbol table as "@0".

$GLOBALGET Function

The $GLOBALGET function takes a single argument:

$GLOBALGET(VAR)

where VAR is any valid identifier except a packed-data name.

Evaluation of the $GLOBALGET function causes the following actions:

1) The $G reserved word is examined. If undefined or nil, go to step 2. Else, for each field $G[1] through $G[$G[0]-1], append subscripts to VAR (the argument to the function), with each new subscript named as the name stored in the corresponding field $G[i].

2) Take the name as altered by step 1 and search the local or global data base for the physically next name.

3) If there is no next-name, return <nil> as the value of the $GLOBALGET function and set the $G reserved word to <nil>.

   If there is a next-name and that name is not subordinate to VAR (ie, if the next-name is not a direct descendant from VAR in the tree-structured database), return <nil> as the value of the function and set the $G reserved word to <nil>.

   If the next-name is subordinate to VAR, take each subscript of the next-name which does not have a corresponding subscript level in VAR, and set $G[1] thru $G[n] equal to the name of each such subscript. Additionally, store the value stored at the next-name in $G[n+1], and set $G[0]=n+1. Return $G[0] as the value of the function. Therefore:

   VAR($G[1],...,$G[$G[0]-1])=$G[$G[0]] and $G(VAR)=$G[0]

For example, if the following nodes are defined:

```
^A(1) = "Z"
^A(1,1) = "Y"
^A(1,1,2) = "X"
^A(1,2) = "W"
^A(2) = "V"
```

Then, successive evaluations of $G will produce:

Command(s)	Result
S $G,A=$G(^A)	A=2, $G[0]=2, $G[1]=1, $G[2]="Z".
S A=$G(^A)	A=3, $G[0]=3, $G[1]=1, $G[2]=1, $G[3]="Y".
S $G[3]=9,A=$G(^A)	A=3, $G[0]=3, $G[1]=1, $G[2]=2, $G[3]="W". Note that ^A(1,1,2) was skipped by SET $G[3]=9.
S A=$G(^A)	A=2, $G[0]=2, $G[1]=2, $G[2]="V".
S A=$G(^A)	A=<nil>, $G=<nil>.

CHAPTER 10: STRING MANIPULATION

10.1 String Manipulation Operators

In section 6.5, Operators, two string-manipulation operators were presented: the concatenation operator (_) and the pattern-matching operator (?).

While outlined in detail in the section referred to above, these operators are included here for the sake of completeness.

The concatenation operator combines two strings into a single string by appending the expression on the RHS of the operator to the expression on the LHS.

The pattern matching operator tests conformity of a string on the LHS of the operator to a pattern on the RHS of the operator.

10.2 String Manipulation Functions

$LENGTH Function

The $LENGTH function takes a single argument:

$LENGTH(EXP)

and returns the number of characters in the value of EXP as its value.

For example:

Example	Result
$L(1=0)	Since 1=0 is FALSE, the value of 1=0 is <nil>, which contains zero characters. Therefore, $L(1=0)=0.
$L("ABC")	Three (3).
$L($L("ABC")=3)	Since $L("ABC")=3 is TRUE, the value of that expression is RUBOUT, which contains one character. Therefore, $L($L("ABC")=3)=1.

$EXTRACT Function

The $EXTRACT function is used to break strings down into substrings, and takes one, two, or three arguments:

```
    $EXTRACT(EXP1)
or, $EXTRACT(EXP1,EXP2)
or, $EXTRACT(EXP1,EXP2,EXP3)
```

The action of the function depends on the number of arguments:

Number of Arguments	Meaning
one	Return the first character of EXP1.
two	Evaluate EXP2 as a number. Return the EXP2'th character of EXP1.
three	Evaluate EXP2 and EXP3 as numbers. Return the EXP2'th-thru-EXP3'th characters of EXP1 (inclusive).

If EXP2 is greater than $L(EXP1), or if EXP3 is less than EXP2, the value of the function is <nil>. If EXP3 is greater than $L(EXP1), EXP3 is treated as being equal to $L(EXP1).

For example:

Example	Result
$E("ABC")	"A"
$E("ABC",0,299)	"ABC"
$E("ABC",4)	<nil>
$E("",0)	<nil>
$E("ABC",2,1)	<nil>

$PIECE Function

The $PIECE function is used to extract a substring that is delimited by some punctuation or alphabetic characters.

The $PIECE function exists in several forms:

$PIECE(ARGS)

In the presentation below, the 0'th occurrence of a delimiter in a string is always the beginning of the string. ARGS may be as follows:

Argument(s)	Meaning
EXP1:EXP2	Evaluate EXP2 as a number. If not a positive integer, use one (1). Return as a value the characters between the EXP2-1'th and EXP2'th colon in EXP1. The following characters may be used as delimiters instead of colon: space, comma, and semi-colon; in which case that character is used instead of colon in the argument to the function.
EXP1:EXP2,EXP3	Evaluate EXP2 and EXP3 as numbers. If either is not a positive integer, use one (1). Return as a value the characters between the EXP2-1'th and EXP3'th colon in EXP1 (exclusive). The following characters may be used as delimiters instead of colon: space, comma, and semi-colon.
EXP1,EXP2,EXP3	Evaluate EXP2. If a number, this form is equivalent to the previous form using comma as a delimiter. If EXP2 does not evaluate to a single character, give an error. Else, EXP2 must be a single alpha or punctuation character. This is the delimiter character. Return as a value the characters between the EXP3-1'th and EXP3'th occurrence of the delimiter character in EXP1.
EXP1,EXP2,EXP3,EXP4	Evaluate EXP2 as a delimiter character. If a number or not a single alpha/punctuation character, give an error. Return as a value the characters between the EXP3'th and EXP4'th occurrence (exclusive) of the delimiter character in EXP1.

$P(C 1) means use space as a delimiter

For example:

Example	Result
$P("ABC,DEF,GHI",2)	"DEF". (First Form)
$P("ABC;DEF;GHI";2,99)	"DEF;GHI" (Second Form)
$P("ABC-EFG","-",1)	"ABC" (Third Form)
$P("ABCDEFEDCBA","B",2,3)	"CDEFEDCBA" (Fourth Form)

SET $PIECE

$PIECE may be explicitly SET, so that the text between delimiter characters may be altered without breaking down the entire string and putting it back together.

Each of the four forms of $PIECE that are outlined above may be SET:

```
S $P(VAR:EXP2)=DATA
S $P(VAR:EXP2,EXP3)=DATA
S $P(VAR,EXP2,EXP3)=DATA
S $P(VAR,EXP2,EXP3,EXP4)=DATA
```

In each case, the number of occurrences of the delimiter character in the original string are counted. If there are fewer than left-index-minus-one (the left-index is EXP2 in forms one and two, and EXP3 in forms three and four) such occurrences, sufficient delimiter characters are appended to the original string.

If no right-index is specified (the right-index is EXP3 in form two, and EXP4 in form four; it is not present in forms one and three), then the piece specified by the left-index is replaced by the new data. If a right-index is specified, then the pieces specified by the left- and right-indexes are replaced by the new data.

For example, if A="ABC:DEF:GHI:JKL", then:

Example	Result
S $P(A:2)="X"	"ABC:X:GHI:JKL"
S $P(A:2,4)="X"	"ABC:X"
S $P(A,"-",3)="XYZ"	"ABC:XYZ:GHI:JKL--XYZ"
S $P(A,":",1,2)	":GHI:JKL"

$FIND Function

The $FIND function is used to locate the position of a substring within a string.

The $FIND function takes either two or three arguments:

$FIND(EXP1,EXP2{,EXP3})

The action of the function is as follows:

1) If EXP2 evaluates to <nil>, return one (1) as the value of the function. This is the only circumstance in which $FIND returns 1.

2) If EXP3 is present, evaluate it as a number. If not present, or if it does not evaluate to a positive integer, use one (1).

3) Starting at the position in EXP1 that was determined in step 2 above, locate the string EXP2 in the remainder of EXP1.

   If not found, return <nil> as the value of the function.

   If found, return the position in which the right-most character of EXP2 was found in EXP1, PLUS ONE, as the value of the function.

For example:

Example	Result
$F("ABCDEF","BC")	Four (4).
$F("ABCDEF","BC",3)	<nil>.
$F("ABCABC","A",2)	Five (5).
$F("ABCADCAEC","AC")	<nil>.

$INVERSE Function

The $INVERSE function is used to invert a string, so that its first character becomes its last, and vice-versa.

The $INVERSE function takes a single argument:

$INVERSE(EXP)

and returns the inverse of the string as its result. The inverse of the <nil> string is the <nil> string.

For example:

Example	Result
$I(123)	321.
$I("MADAM IM ADAM")	"MADA MI MADAM".

$CHARACTER Function

The $CHARACTER function is used to map ASCII codes (see 16, Appendix I - ASCII Codes) into their corresponding characters.

The $CHARACTER function takes a single argument:

$CHARACTER(EXP)

and returns as its value the character whose ASCII code is EXP. If EXP is greater than 128, the $CHARACTER function returns <nil>.

For example:

Example	Result
$C(7)	BELL.
$C(65)	"A".
$C(127)	RUBOUT.

$HASHCODE Function

The $HASHCODE function is the inverse of the $CHARACTER function, and maps characters into their corresponding ASCII codes.

The $HASHCODE function takes either one or two arguments:

$HASHCODE(EXP1{,EXP2})

The action of the function is as follows:

1) If EXP2 is present, it is evaluated as a number. If not present, or if it is not a number, use one (1).

2) Return the ASCII code of the character in EXP1 which is pointed to by EXP2. If there is no such character, return zero (0).

For example:

Example	Result
$H("ABC")	65.
$H("ABC",2)	66.
$H("ABC",4)	0.

$XCLUSIVEOR Function

The $XCLUSIVEOR function is used to generate and accumulate longitudinal redundancy check values for inter-processor communications.

The $XCLUSIVEOR function takes either one or two arguments:

$XCLUSIVEOR(EXP1{,EXP2})

The action of the function depends on the number of arguments:

Number of Arguments	Meaning
one	Accumulate and return the value obtained by XORing the ASCII codes of each character in EXP1. XOR (exclusive-or) is a bit-wise operation that is equivalent to: A <XOR> B = (A <or> B) - (A <and> B) for bit-wise and/or.
two	Evaluate EXP1 and EXP2 as numbers, and return the result of: EXP1 <XOR> EXP2.

For example:

Example	Result
$X("ABC")	64.
$X(64,63)	127.
$X(6,10)	12.

CHAPTER 11: MISCELLANEOUS COMMANDS

11.1 XECUTE Command

The XECUTE command allows MIIS code to be executed when it is someplace other than loaded in a program buffer.

The XECUTE command takes a list of arguments:

XECUTE arglist

Each argument is an expression whose value is executable MIIS code.

The code is executed from the user's stack, which means that it is possible to LOAD programs in the course of the XECUTE command.

Any programs so loaded will remain the currently loaded program until another program is loaded. It is therefore necessary for XECUTE commands that appear in programs and that LOAD other programs to re-load the original program before they terminate.

A CALL, DO, or GOTO command that appears inside an XECUTE argument will be executed, with the following termination conditions:

Command	Terminating Condition
CALL	A QUIT will return to the point inside the XECUTE after the CALL command.
DO	A QUIT will return to the point inside the XECUTE after the DO command.
GOTO	A QUIT will terminate the XECUTE, and return control to the command immediately following the XECUTE.

OVERLAY commands may not appear inside XECUTE arguments.

For example, consider the following code:

```
PGM1    X "L PGM2 P  L PGM1 Q"
        X "W 1 D A C PGM2 G B W 4"
        W 5 Q
        ;
A       W 2 Q
        ;
B       W 6 Q

PGM2    W 3 Q
```

The result of CALL PGM1 will be:

1) PGM2 is printed on the principal device.

2) The following digits are output in this order: 12365.

11.2 JOB Command

The JOB command is used to initialize parallel processes by starting another MIIS job. The JOB command may not be executed in direct mode, but must be executed as part of a stored program line.

The JOB command takes a list of arguments:

```
JOB arglist
```

where each argument on the list is a program name. The programs named in the JOB command must be accessible via signing-on in user mode (see section 2.3, Installing and Accessing MIIS), which means that each such program must be stored with a "#" as the last character in its name.

As with signing on, the name of the program as it appears as an argument in the JOB command must be without the "#". The "#" is appended automatically by the system.

If no free partitions are available when a JOB command is executed, the job issuing the JOB command is placed on the resource queue to wait until a partition is freed.

The newly started job will copy the following from the starting job:

1) $QUEUE and $RESOURCE reserved word values.
2) Principal Device.
3) Namespace names.
4) $G reserved word.
5) Entire local symbol table, including %-variables.

The newly started job will be in user-mode, and will have no devices assigned and no globals opened. Its IF-SWITCH is initialized FALSE.

11.3 HALT and HANG Commands

The HALT and HANG commands begin with the same letter. They are distinguished by the presence of arguments.

HALT Command

The HALT command is used to terminate MIIS jobs.

The HALT command takes zero arguments:

```
HALT
```

and has the following effect:

1) If the job owns its principal device, the message "GOOD BYE" is printed on the principal device.

2) All owned devices are released.

3) All opened globals are closed.

4) The job is terminated, releasing the partition for another job.

HANG Command

The HANG command allows jobs to suspend themselves for a period of time.

The HANG command takes a list of arguments:

```
HANG arglist
```

where each argument in the list is an expression that evaluates to a number. The effect of the command is to make the job clock-hung until that number of seconds has elapsed, thereby suspending its processing during that period.

The command HANG 0 has a slightly different effect. The job is simply swapped-out as though its slice had expired, and is placed back on the end of the CPU wait queue (see sections 14.1, Different Job States, and 14.5, The Timesharing Executive).

11.4 BREAK and GO Commands

BREAK Command

The BREAK command is used to stop a process in a restartable manner, so that the programmer can examine the state of the machine and resume processing.

The BREAK command takes zero arguments:

```
BREAK
```

and when executed causes the following actions:

1) If the job is in user-mode (see 2.3, Installing and Accessing MIIS), generate a ?BREAK error and sign the job off.

2) If the job is in programming-mode, generate a ?BREAK error and return to direct-mode. The condition of the user stack is preserved, as is the program pointer (that points to the character of code being executed).

Any MIIS ?ERROR that is generated while the job is in direct-mode after a ?BREAK error will cause the user stack to be re-initialized.

GO Command

The GO command is an auxiliary to the BREAK command, and causes processing to resume at the instruction immediately following the BREAK command.

BREAK commands may be nested, with corresponding GO commands restoring processing at the instruction immediately following the most-recently issued BREAK command.

The GO command may only be issued in direct mode (to distinguish it from the GOTO command), and takes zero arguments:

```
GO
```

If the GO command is issued without a corresponding BREAK command, a ?STACK error is generated.

Consider the following terminal session:

```
?P A
A       W !"MESSAGE ONE" B  W !"MESSAGE TWO" Q

?D A

MESSAGE ONE?BREAK
A       W !"MESSAGE ONE" B  W !"MESSAGE TWO" Q

?G

MESSAGE TWO
?
```

11.5 Comments in Programs

To insert a comment in a MIIS program, place a semi-colon (;) where a command name might appear.

All text to the right of the semi-colon to the end of the line is then ignored by the interpreter.

Such commentary does, however, occupy space in the program buffer. There is therefore a trade-off between the illustrative value of the comment and the amount of space used by each program.

For example:

```
A       F I=1:1 Q:'A(I) W !,A(I) ;THIS IS A COMMENT
        ;AS IS THIS
B       ;AND THIS
```

CHAPTER 12: MISC. FUNCTIONS AND RESERVED WORDS

12.1 $TIME, $CONSOLE, and $KONSTANT

$TIME Reserved Word

The $TIME reserved word is a count of the number of seconds that have elapsed since 12:00:01 AM, January 1, 1976. It is initialized by the STD utility program, and incremented by MIIS each time a second goes by.

$TIME#86400 gives the number of seconds since midnight on the current day. $TIME#86400/3600 counts the hours; $TIME#86400#3600/60 counts the minutes in the current hour; and $TIME#86400#3600#60 (which is equivalent to $TIME#60) counts the seconds in the current minute.

The following line of code prints the current time of day in AM/PM format:

```
S A=$T#86400/60 W A/60+11#12+1,":",A/10#6,A#10,A>720&"PM"!"AM":1
```

Note that $TIME/86400 is the number of days since January 1, 1976. Note also that there were 64223 days from March 1, 1800 to January 1, 1976. Therefore, the number of days between March 1, 1800 and the present is given by $TIME/86400+64223. This number is the internal date format used by the MIIS date utilities and WRITE EXP:Dn syntax.

If today is May 9, 1980, WRITE $T/86400+64223:D8 outputs 05/09/80.

$CONSOLE Reserved Word

The $CONSOLE reserved word expresses the state of the computer's console switches.

For computers with a front switch panel (Eclipse C/350, PDP 11/70, IBM Series I), some MIIS assemblies allow the state of the console switches to be established with the front panel switches:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
W			S			D			C			H	%	K	D
R			Y			W			N			L	M	B	D
T			S			N			S			T	X	G	T

(The significance of these switches is explained in chapter 19, Appendix IV - Console Switches and DDT).

In this event, the switches are physically raised and lowered to set and reset. The value of the $CONSOLE reserved word is the decimal number obtained by treating the switch register as an octal (binary) number and converting. For example, if the SYS and KBG switches are raised, $CONSOLE=4096+2=4098.

Some computers (Eclipse S/140, PDP 11/23) do not have physical switches. In addition, some MIIS assemblies do not support physical switches even on computers that have them. In this event, MIIS allows a syntax in which $CONSOLE(SWITCH) may be explicitly SET.

This may be done at the MIIS level by:

SET $CONSOLE(SWITCH #)=T/F

where TRUE means SET and FALSE means RESET. The switch numbers are given in the above diagram (0-15). The job executing the command must be assigned to the computer's console terminal, and must have the console as its principal device. Setting or resetting the WRT switch requires that the view buffer be owned. Setting or resetting switch numbers 12 (HLT) or 15 (DDT) with SET $CONSOLE is illegal. In order to test the state of a particular switch, $CONSOLE is tested for the presence of that bit in its binary equivalent.

There is also a utility program called SWT that sets and resets switches from the computer's console terminal. This is documented in the MIIS utilities manual.

For systems that require SET $CONSOLE as the mechanism for setting the switches, the switches may also be set using a special syntax in DDT (see section 19, Appendix IV - Console Switches and DDT). DDT itself is invoked by hitting CTRL/S at the console terminal, and is terminated by CTRL/Q. This is the only way available to set the system HLT switch.

$KONSTANT Reserved Word

There is a table in MIIS called the in-core table (or the constants table) which contains a set of constants that pertain to a particular MIIS assembly.

The address of this table is given by the value of the reserved word $KONSTANT.

Some of the important items in this table are listed below:

Reference	Name	Meaning
$W($K)#8	MAC	Type (vendor) of CPU.
$W($K)/8	DTY	Type of disk drives.
$W($K+1)	JF	Address of first job.
$W($K+2)	JMX	Maximum number of jobs.
$W($K+4)	VBF	Address of beginning of view buffer.
$W($K+6)	PWT	Address of beginning of password table.
$W($K+10)	DMX	Maximum device number.
$W($K+28)	DBC	Binary coding of online disks.

These constants may be edited with the utility program CON, and listed with the utility program COP.

12.2 $X-COORDINATE and $Y-COORDINATE

The reserved words $X and $Y are maintained by the operating system and count certain input/output actions.

$X is set to zero every time a WRITE ! (CR/LF) or WRITE # (FF) is executed, and is incremented by one each time a printing character (ASCII space through ASCII tilde) is output with either WRITE or WRITE-STAR.

$X takes values from 0-255 inclusive, and wraps around to zero.

$X does not change when keys are struck on a terminal.

$Y is set to zero each time a WRITE # (FF) is executed, and is incremented by one each time a WRITE ! is executed. $Y takes values from 0-255 inclusive, and wraps around to zero.

$Y is not incremented by striking the line-delete key (which outputs a ?/CR/LF), but is incremented by other direct-mode input.

12.3 $IO, $STORAGE, $PASSWORD, $QUEUE, $RESOURCE

$IO Reserved Word

The $IO reserved word has the number of the job's current I/O device as its value (see section 7.2, ASSIGN and UNASSIGN Commands). If there is no current I/O device, the $IO reserved word is equal to the job's principal device number.

$STORAGE Reserved Word

The $STORAGE reserved word contains the number of free bytes in the job's partition. This is obtained by:

2048-(# bytes in local symbol table)-(# bytes in user stack).

$PASSWORD Reserved Word

When passwords are created with the utility program PW, each password is assigned a number. The $PASSWORD reserved word is equal to the password number (see section 2.2, Passwords and their Implications) to which the job is currently attached.

There are two ways in which a job may be attached to a password:

1) By signing on directly into that password, using its three-letter code.

2) By explicitly SETting $PASSWORD to a password number.

The utility program PWP can make a password ineligible for transfers to and from other passwords. Passwords so declared may not update %-globals.

$QUEUE Reserved Word

The $QUEUE reserved word takes values from 0-7 inclusive, and represents the number of the CPU wait queue in which the job circulates.

$QUEUE is explicitly SET to the appropriate value.

The eight different wait queues are serviced in the order of their number. No jobs in queue N will run if there are any jobs in queues 0-(N-1) that are requesting service, except that every sixty-four (64) seconds, all CPU-queued jobs will run for one slice regardless of which wait queues they are in.

When a job signs on, and each time the job returns to direct mode, its $QUEUE reserved word is set to two (2).

$RESOURCE Reserved Word

The $RESOURCE reserved word takes values from 0-3 inclusive, and indicates how long the job will run once it gets an opportunity to run (how often the job runs is determined by its $QUEUE reserved word).

$RESOURCE is explicity SET to the appropriate value, and controls the number of "ticks" of the real-time clock that will be allowed during a job's execution before the job is swapped out. The total time that the job runs before it is swapped out is called a "slice". The real-time clock ticks sixty times per second in 60hz countries.

$RESOURCE also controls the number of consecutive logical disk references that require physical disk that the job will be allowed to make before the job is swapped out.

$RESOURCE	Ticks per Slice	Logical Disk References
0	2	2
1	4	4
2	6	6
3	8	8

When a job signs on, and every time the job returns to direct mode, the $RESOURCE reserved word is set to zero (0).

The management of swapping is further discussed in section 14.5, The Timesharing Executive.

12.4 $MULTI-PRECISION and $RANDOM

$MULTIPRECISION Function

The $MULTIPRECISION function is used to perform floating-point arithmetic.

$M takes either one or two arguments:

$MULTI(FEXP+FEXP-FEXP*FEXP/FEXP{,FMT})

where the first argument consists of a number of floating-point numeric expressions combined with arithmetic operators, and the second argument is an output field-width specifier. It is important to note that parenthetical subexpressions are illegal inside $M.

The precision retained for addition, subtraction, and multiplication is the same as if the operation had been performed with paper and pencil. Accuracy is maintained for all operations to sixteen (16) significant figures; any additional figures required to the left of the decimal point as low-order place-holders are returned as zeros.

The optional formatting operator is specified as:

n.m

where "n" and "m" are literal integers. The result is returned with "n" places to the left of the decimal point and "m" places to the right of the decimal point. The result is padded with leading spaces and trailing zeros to achieve the format.

$M will return the right result in the wrong format rather than the wrong result in the right format.

Some examples (where "_" represents a space character):

Example	Result
$M(1.2+3.40)	4.60
$M(1.2+2*3.5)	11.20
$M(1.2-A+B)	0 if A=".2" and B="-1".
$M(1+1/3)	.6666666666666667
$M(1+1/3,.2)	.66
$M(10/4)	2.5
$M(100000000000000000000/3)	3333333333333333000
$M(1.2+3.49,3.1)	_4.6
$M(10*6.5,.1)	65.0
$M(0,3.2)	__0.00
$M(-1,3.0)	_-1

Note that $M does not round when fitting a result into a format. Rounding may be accomplished as in this example:

```
$M(2/3+.005,.2)=.67
```

$RANDOM Function

The $RANDOM function generates random numbers.

The function takes a single argument:

```
$RANDOM(EXP)
```

where the expression evaluates to a number. The random number is generated between zero and EXP-1 (so that $R(0)=$R(1)=0).

The expression EXP is taken modulo 65536 in determining the range of the generated random number.

CHAPTER 13: DATA STORED ON THE DISK

13.1 Physical Nature of the Disk

Physical Description of the Disk

In general, a "disk" is some number of magnetic surfaces that rotate at high speed past an equal number of recording/playback heads. The record/playback heads are usually part of the disk "drive" on which the surfaces are mounted. The surfaces themselves are part of the disk "pack". Disk packs are often removable, so that different packs may be placed on the same drive at different times.

Disks are manufactured by a wide variety of vendors, and exhibit a large range of capacities, operational differences, and performance characteristics. There are, however, some basic similarities in the geometrical architecture of the various disk units.

Disk units are organized with a cylindrical coordinate scheme that enables any physical point on the disk pack to be identified by a set of coordinate values. A typical multi-disk configuration requires a four-dimensional representation of its geometry:

Coordinate	Meaning
DRIVE #	The spindle (disk drive) on which the pack is mounted.
CYLINDER #	The radial distance from the center of the pack. Two items in the same cylinder may be picked up without an intermediate movement of the heads.
SURFACE #	The distance from the bottom of the pack. Two items on the same surface may be picked up by the same head, although movement of the heads may be required.
SECTOR #	The angle from top-dead-center at which the data is stored. Two items in the same sector are picked up at the same point in the disk's rotational period.

Each unit specified by such a set of four coordinates is called a "block". Each block has, in addition to its drive/cylinder/surface/sector address, an absolute block number which may be computed from its four-coordinate address:

BLK=DRIVE*w+(CYL*x)+(SUR*y)+SEC

where w, x, and y depend on the particular hardware.

Disk blocks contain a variety of information that is used by the hardware to do its job. This information is set up by the vendor's formatter program, which must be run on each disk pack before it can be used for MIIS.

In addition to this hardware information, each disk block contains space for 512 characters of data.

Certain disk blocks are used for special purposes:

Block(s)	Purpose
0-127	Used to store the image of the operating system for bootstrapping.
1	Used to store the label block that identifies the disk pack. Note that this block corresponds to bytes 512-1023 of the operating system image. This is where the view buffer is stored, so that on initial boot, the view buffer will contain the label block.
128-n	Used to store the bit map blocks (more on these later). The number "n" depends on how many disk blocks are present in the configuration.
n+1,n+2	The program and global directory blocks, respectively, for the system password.

Structure of Disk Blocks

There are seven different kinds of disk blocks that are used by MIIS:

1) Program pointer-level blocks.
2) Program data-level (level 0) blocks.
3) Global pointer-level blocks.
4) Global data-level (level 0) blocks.
5) Map blocks.
6) Operating system image blocks.
7) Label block (as noted above, a special case of (6)).

As noted above; each of these types of blocks contains space for 512 bytes of accessible information.

The first four types of disk blocks are used to store and to locate program and global information. The internal structures of these blocks are only rarely of concern to the applications programmer. Details of how these blocks are constructed are presented in 17, Appendix II - Disk Block Structure. The logical scheme of how these blocks interact (are chained together) is presented in the next section, 13.2, Physical Storage of Globals and Programs.

The latter three types of disk block (numbers 5-7) are not of direct concern to the applications programmer; the information they contain is used behind the scenes. Map blocks are used by MIIS to record which disk blocks are in use at any given moment. The operating system image blocks are used to store a copy of the system so that it may be reloaded conveniently. The label

block is used by the utility programs to ensure that the correct disk packs are mounted on the correct disk drives.

13.2 Physical Storage of Globals and Programs

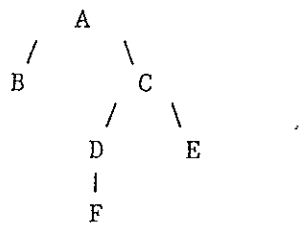
This section describes how MIIS global and program data is physically stored on the disk, and outlines some of the dynamics behind lookups and updates.

The important point to remember is that the logical relationships among the data are independent of the physical representation of the data. The applications programmer may view the database as a black box that can store and unstore the information he specifies.

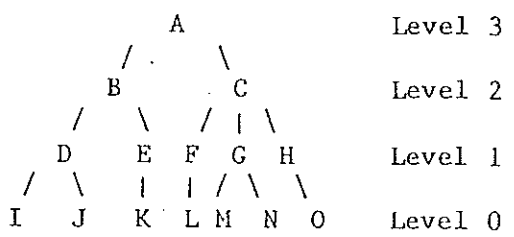
Balanced Trees

Balanced trees (B-trees) are a particular type of tree structure in which the length of the path from the root (top) of the tree to any bottom node of the tree is the same as that to any other bottom node. This implies that the tree is balanced vertically, with well-defined levels:

Non-Balanced Tree:



Balanced Tree:



MIIS uses the B-tree to enable quick access to particular pieces of information. The access is made by the name of the information.

All MIIS global information is actually stored in what may be described as a linear (sequential) file of disk blocks that are chained together with "continuation pointers". This chain of disk blocks, each block containing between 10 and 512 characters of (name,value) pairs, represents the bottom level (sequence of lowest-level nodes) of the balanced tree structure illustrated above. For example:

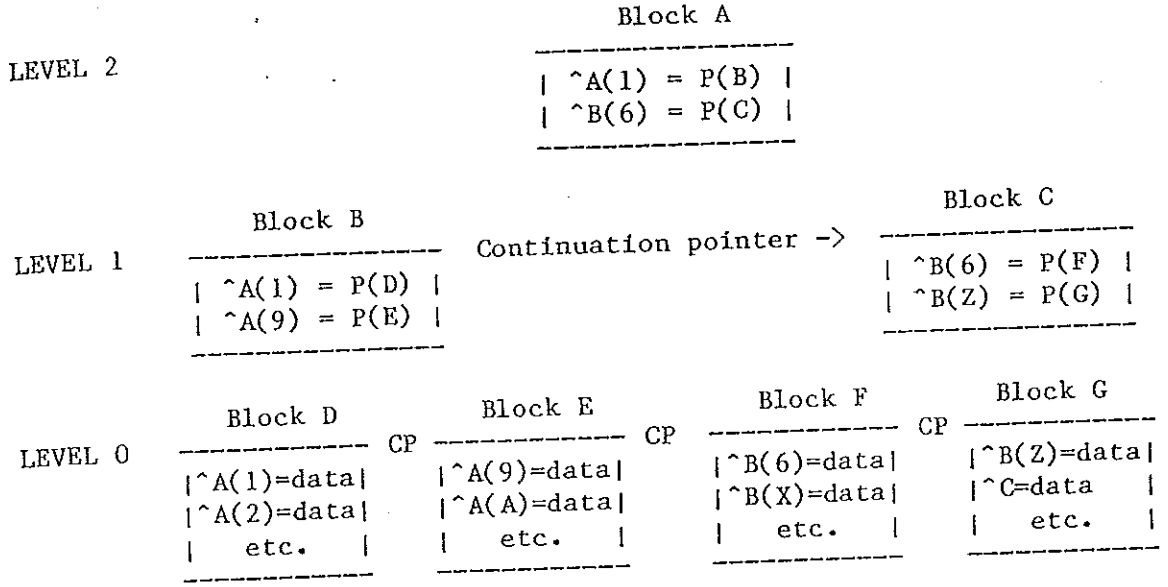
I -> J -> K -> L -> M -> N -> O

The (name,value) pairs are stored in collation sequence of the name within each block, and the names in each block collate before the names in its continuation block. The upper levels are also chained sequential files, and are name-locating pointers into the successively lower levels.

All blocks at level zero contain (name,value) pairs. All blocks at higher levels contain (name,pointer) pairs, where the pointer is the number of the block at the next-lower level that immediately descends from the current block, and the names in pointer-level blocks match or precede the first name

in the block that is pointed to.

The following is a picture of how a simple B-tree is organized. P(n) represents a pointer to block "n":



In order to locate ^B(1), for example, we would determine from block A that names between ^A(1) (inclusive) and ^B(6) (exclusive) are in block B. From block B, we determine that names after ^A(9) are in block E. Block E is a level-zero block, so if ^B(1) exists, it will be found there.

How the B-tree Grows

The dynamics of the growth and pruning of the B-tree are more complicated than those of lookups.

Each password in a MIIS system has its own B-tree for storage of global information. The root of this tree is the password's global directory block.

When the password is initialized, the B-tree consists of an empty level zero global directory block with a single entry: the <nil> name with a <nil> value.

As (name,value) pairs are inserted into the password's database, the level zero global directory block becomes filled. When it is so full that the next (name,value) pair does not fit inside the 512-byte block, the contents of the block are split. Two new blocks are allocated (this process is described later), and each half of the old block is put in a new block. The global directory block then becomes a level one block, containing pointers to the first names in each of the two new level zero blocks.

As the new level zero blocks become filled, they are split in turn, and entries are made in the level one block to point to the first names in each level zero block. Eventually the level one block becomes filled with pointers, and IT is split. Two level one blocks are created, and the old

level one block (which is still the global directory block) becomes a level two block with pointers to each of the level one blocks.

When deletions are made from the database, the appropriate (name,value) pairs are removed from their level zero blocks. If the level zero block is only partially emptied by removing the (name,value) pair, there is no further action. If the level zero block is completely emptied, it is removed from the level zero chain, and the corresponding pointer is removed from the level one block that points to it. If the level one block is emptied by removing that (name,pointer) pair, the level one block is removed from its chain and deleted from its superior level two block, and so on.

In no case however, can the number of levels of the database be reduced by deleting entries. Deletion (KILLing) of all the nodes in the database would result in a global directory block with a single (name,pointer) pair, using the <nil> name. There would be only a single block at each level, each with a similar (<nil>,pointer) entry. The level zero block would have only a <nil> name entry with a <nil> value.

Efficiency of Storage

It was mentioned above that when a block becomes full, its contents are divided between two blocks. The rules of this "splitting" process affect the average number of bytes in each disk block that are actually used to store global data.

There are two particularly interesting places to divide the data in the meiotic process: at the insertion point of the (name,value) pair that caused the block to split, or in the middle of the disk block regardless of where the new data is inserted.

The first plan will clearly provide the most efficient storage for sequential (in collation order of name) insertions to the database (because the insertion point of the splitting entry will always be at the end of the disk block, leaving that block full, and the next block will be filled by the subsequent sets). On the other hand, random insertions result in less than optimal efficiency under this scheme.

The second plan (splitting in the middle) will give at least 50% efficiency for both types of insertions. In practice, it may be shown that this plan averages 50% efficency for sequential insertions and 70-80% efficiency for random insertions.

It therefore seems that some sort of hybrid approach is appropriate. This is implemented in two ways:

1) The STRUCTURED SET syntax will always insert data in collation sequence. Sets to the database using a STRUCTURED SET command therefore use the split-at-the-insertion-point algorithm.

   Other insertions to the database use the split-in-the-middle rule, except:

2) On IBM SERIES 1 and DEC PDP-11 computers, MIIS keeps a record in each disk block of which node was last inserted in the block. If a subsequent insertion immediately follows the last-inserted entry, the new insertion follows the split-at-the-insertion-point algorithm; otherwise, the block is split-in-the-middle.

Allocation of New Disk Blocks (Using Map Blocks)

When MIIS needs a new disk block to accomodate the overflow of an existing block, it must ensure that the new block it uses is not already in use in another structure. Since the data in the blocks looks the same whether or not the block is part of a structure, there must be some method for finding free blocks.

This is what map blocks are for. A disk block contains

512 bytes/block x 8 bits/byte = 4096 bits/block.

Map blocks are a collection of blocks whose bits are used to correspond to free or not-free disk blocks. When a system is delivered, certain blocks are already in use: blocks 0-127 which store the operating system image, and the map blocks themselves. As further blocks are allocated, the corresponding map block bits are zero'ed.

Groups of 4096 blocks that are spanned by map blocks are logically termed "maps". Many of the utility programs section the disk into these logical constructs. Map zero is the first 4096 blocks on the first disk drive.

Since map blocks are so crucial to the safe operation of the MIIS system (if one goes bad, data which was written to disk will be overwritten and lost), they are written to the disk immediately upon being updated, and, unless already present in the special disk buffer reserved for map blocks, are brought in from the disk each time a new block is allocated. However, it would be very wasteful to bring in a map block only to find out that there are no free blocks in that map.

For this reason, MIIS maintains a table called the "free map" in local memory. The free map contains one bit per map block, indicating whether that map contains any free blocks.

When a new block is to be allocated, MIIS looks at the free map for the first not-full map block between the password's global directory and the password's (perhaps implicit) bound. That map block is then read into memory, and the

first free block in the map is allocated.

Note that an insertion into the database may require as many new blocks as there are levels in the database (this could happen if the blocks split at every level as a result of the insertion). If there are not enough free blocks, MIIS will abort the insertion in a legal state and return a ?SPACE error.

Program Storage

Programs are stored in chained lists of disk blocks, using all 512 characters of each disk block to store program text. In order to access the header block of the chain that contains a particular program, MIIS looks up the program name in a B-tree structure that is headed by the password's program directory block.

The detailed structure of program storage blocks is presented in 17, Appendix II - Disk Block Structure.

13.3 File Protection: OPEN, CLOSE and $UPDATE

In a timesharing environment, it is possible that two users might be updating the same file at the same time, with unexpected results.

For example, suppose users A and B are executing the following code at the same time:

```
SET A=^CNT+1,^CNT=A
```

and user A gets as far as "S A=^CNT+1" and gets swapped out, and user B gets swapped out at the same point in the code. Then ^CNT gets incremented only by one, and users A and B get the same value in the variable A. This is clearly not the intent of the code.

For this reason, MIIS has the OPEN and CLOSE commands. These are used to enable jobs to communicate with one another and prevent timesharing conflicts.

OPEN Command

The OPEN command is used to create an entry in the system's open table that indicates that a particular job has OPENed a global node.

The OPEN command takes a list of arguments:

OPEN arglist

where each argument in the list is one of the following:

Argument	Meaning
GVAR	Try to OPEN the named global variable. If possible, create the entry in the OPEN table. If not possible, place the job on the resource queue until it is possible to perform the indicated OPEN.
GVAR:EXP	Try to OPEN the named global variable. If possible, create the entry in the OPEN table and set the IF-SWITCH FALSE. If not possible, set the IF-SWITCH TRUE and give up.

There are four reasons that it might not be possible to perform a given OPEN command:

1) Because the OPEN table is full.

2) Because another job has the same global node already opened in the same password.

3) Because another job has a direct descendant of the global node opened in the same password.

4) Because another job has a direct parent of the global node opened in the same password.

The utility program %OPT will print a list of the entries currently in the open table, and will indicate how full the open table is.

OPENing a global name that is already opened by the same job results in a duplicate entry being created in and taking up space in the open table.

It should be noted that the OPEN command makes no reference to the disk unless expressions that form a part of the name of the argument to the OPEN command contain global references. It should also be emphasized that use of the OPEN command is conventional only, and that no automatic protection of files takes place - only protection against other OPENs.

For example, after OPEN ^A(1,2), other jobs:

Can Open	Can't Open
^A(2)	^A
^A(1,3)	^A(1)
^A(1,3,2)	^A(1,2)
^A(12)	^A(1,2,3)

CLOSE Command

The CLOSE command removes entries from the open table.

The CLOSE command takes either zero arguments or a list of arguments:

```
    CLOSE
or, CLOSE arglist
```

The CLOSE command without any arguments removes any entries from the open table that were established by the current job.

If arguments are present, each is a global variable name. If there is such an entry in the open table for the job, the first occurrence of the entry is removed. Otherwise, no action is taken.

After a CLOSE command is executed, all jobs on the resource queue are cycled through the wait queues.

$UPDATE Function

The $UPDATE function may be used to avoid time-sharing conficts because it is completely evaluated in a single user time slice. Section 9.3, $UPDATE Function contains a detailed description of the function.

CHAPTER 14: THE RUN-TIME OPERATING SYSTEM

14.1 Different Job States

The executive of the MIIS operating system controls which processes are performed at which times. This scheduling algorithm requires that each job exist in one of two states: either "hung" or "queued".

Queued jobs are waiting for their chance to use a system resource, and each queue is internally represented as a list of jobs. Because each such list of jobs has a beginning and an end, it is possible to recognize immediately whether there are any jobs on a specific queue just by looking at the queue's beginning point.

Hung jobs have initiated a process on a system resource, and are waiting for that process to finish. Because hung jobs are not in a list like queued jobs, it is not possible to ascertain immediately which jobs are in a particular hung state.

Queued Jobs

There are six different queued states that MIIS jobs may be in, each waiting for a different logical "resource".

The jobs on the CPU WAIT queue are waiting to do some processing on the CPU. This queue may be thought of as eight separate queues, one for each possible value of the $QUEUE reserved word. Any mention of the CPU queue is really a reference to the appropriate segment of that queue which is specified by the value of $QUEUE. Jobs on the CPU queue circulate as each is serviced in turn.

The DISK queue is a list of jobs that are waiting to access the disk. These jobs are waiting to access the physical disk, or to change data on the logical disk (e.g., the buffer pool). Like the CPU queue, the disk queue is sorted by the $QUEUE reserved word. Jobs on the disk queue return to another state after being serviced.

The FREE queue is a list of jobs that are currently signed off. Whenever the BREAK key is struck on a appropriate terminal or a JOB command is executed, the free queue cycles. If there are no jobs on the free queue, attempts to sign on will result in a "FULL" message, and attempts to JOB a program will wait on the resource queue until a job becomes free.

The RESOURCE queue is a list of jobs that are waiting for one of several events:

1) For a device to become free so that it may be ASSIGNed.

2) For an entry to be removed from the open table so that a global may be OPENed.

3) For a partition to become free so that a JOB command may be executed.

Therefore, after each UNASSIGN, CLOSE, or HALT command is executed, all jobs on the resource queue are appended to the CPU wait queue in order to try again to obtain the appropriate resource. Jobs on the resource queue are sorted in $QUEUE order.

The RUN queue consists of at most one job, and that is the job which is currently active.

The "immediate" queue (called Q-IMMEDIATE) consists of at most one job, and that is the job which has just completed a reference to the physical disk and now wants to make logical use of the data now in memory.

Hung Jobs

A DISK HUNG job is waiting for a block to be transferred from the disk to the buffer pool. There may be only one disk hung job at any given time, although the system snapshot utilities may show more than one.

CLOCK HUNG jobs are waiting for a certain number of seconds to elapse on the real-time clock before they resume processing. Jobs are clock hung because of HANG commands.

INPUT HUNG jobs are performing input commands from terminal devices. The jobs remain input hung until the enter key is hit on the terminal.

OUTPUT HUNG jobs are performing output commands directed at terminal devices. An output hung job is attempting to output more than a buffer-full of text to the terminal.

TRAP HUNG jobs are deemed by MIIS to be in an un-runnable state, and will not circulate on any queue until the job is aborted either by a system reload or with the DEV utility program. Jobs become trap hung by being disk hung, magtape hung, on the run queue or on q-immediate when MIIS is restarted.

14.2 Local Symbol Table and $HASH Reserved Word

All MIIS local symbols share the job's partition with the user's stack and program vector.

The partition is structured so that the symbol table and stack grow toward one another. The total available space is roughly 1995 characters to be shared between the two structures (which implies that the program vector occupies 2048-1995=53 bytes).

The local symbol table may be thought of as a single logical "block" without an access structure imposed on it as with global data. In order to access a particular symbol, it would be necessary to search the entire local symbol table for a matching entry if it were not for the multiple entry points into the symbol table that are maintained by the operating system.

These entry points are maintained in the reserved word $HASH which may be SET to an expression that evaluates to a four-character string. This four-character string must consist of characters that collate after a question-mark (63 decimal) in the ASCII sequence. Each character in the string must have an ASCII code greater than that of the previous character. Thus, "@ABC" is a valid string, as is "XYZa", but "DCBA" is not.

Each job may SET $HASH to its own needs. $HASH is delivered set to "AHOX", and will revert to that state when a job is first signed on; however $HASH is preserved when a JOB command initiates a new process in another partition.

$HASH may NOT appear as part of a MIIS expression, but may only be SET.

Each time a local symbol is referenced, MIIS compares the first letter of the requested symbol with each character in $HASH. This comparison enables MIIS to begin the symbol table search at the $HASH entry closest to but not beyond the point where the requested symbol is stored.

14.3 Program Buffers

The partition contains three things: the local symbol table, the user's stack, and the program vector. The program vector contains a variety of run-time information, including a pointer into the program buffer that contains the currently loaded program.

Program buffers are essentially a separate buffer pool of 2KB blocks, each capable of accommodating a MIIS program. They are used in order to allow improved use of system resources:

1) It is not necessary to have the same number of program buffers as there are jobs on the computer system. If a program buffer is required but not available, the job waits until one is free.

   This can save on space, or existing space may be used for disk buffers.

2) Since programs are not flushed from the program buffer pool until an incoming program forces out the least-recently-used program, it is likely that an unattached program will last for several minutes in the pool. This means that a program can call another (which may be in the pool, thereby not requiring a disk access), run the other program, and return to the original program without fetching any programs off the disk.

Each program buffer contains the following information:

1) The program text itself.

And in the program buffer control blocks (management table),

2) The password number with which the program is associated.

3) Address in memory of program buffer.

4) The number of jobs that are currently attached to the buffer (this is called the "use count").

5) The least-recently-used count (LRU), which is updated each time a job accesses the buffer.

6) A flag that indicates that the program buffer is "locked", and may not be used by any jobs other than the one to which it is currently attached.

When a program is loaded (by the LOAD, CALL, or OVERLAY command), MIIS searches the buffer control blocks for the requested password number, and then looks in the buffer itself to compare the program name. If the program is found and the flag mentioned in number (6) above is not set, the job is linked to that buffer, its LRU is updated, and its use count is incremented.

Otherwise, MIIS finds the buffer with a zero use-count that has the lowest

LRU-count, copies the program from the disk into that buffer, and links the job to that buffer.

If the system is configured with the same number of program buffers as jobs, there will always be a free buffer in the event the requested program is not already in memory. If the system is configured with fewer program buffers than jobs, it is possible that there may not be a free buffer, in which case the job will hang until one becomes available.

Whenever a program is loaded, the program buffer (if any) which is relinquished has its use-count decremented.

Changing and Filing Programs

When the code that is loaded in a program buffer is changed by a programmer, the programmer is provided with his own copy of the program to edit.

This happens as follows: if the programmer is sharing the program with other users, the as-yet-unchanged code is copied into a new buffer, and the programmer is linked to that new buffer. Then, the buffer to which the programmer is linked has its one-user flag (number 6 above) set.

Therefore, any new users wanting to access that program will either link to the still-unchanged copy (if there is one left in the pool) or will bring in a new unchanged copy of the program from the disk.

When the programmer FILEs the program, MIIS looks for a copy of the program in the buffers that is accessible with a LOAD command (there either zero or one such buffer, depending on whether there were other jobs using the program that was modified). If there is such a copy, its flag is set to prevent new jobs from attaching that program buffer. This means that those jobs currently attached to the buffer may finish executing, but that subsequent references will go to a new copy of the changed program as brought in from the disk.

14.4 Global Buffers and Disk Management

It has been noted many times that MIIS deals with the physical interface to data stored on the disk, and that the programmer need only be concerned with the logical relationships between the data.

This section will describe the management of the physical interaction with the disk.

Buffer Pool

The buffer pool is a collection of disk block images that are stored in local memory. Each block image is called a "buffer", and is accessible to any job on the computer that might require it.

In addition to the set of disk block images, there is a management table for the buffers which indicates which physical blocks are stored inside which physical buffers.

This table also keeps track of which physical buffers were most-recently required by a MIIS job by assigning a sequential number to each examination of a buffer, and storing the current value of that number (called the LRU for Least Recently Used) in the entry in the buffer pool table that corresponds to the buffer just examined.

The LRU is used to determine which physical buffers have their blocks overwritten when a new disk buffer is required. This is explained in more detail later.

The main point about disk buffers is that all traffic between the physical disk and the jobs making references to the physical disk take place through the buffer pool. In other words, all program and global blocks that are requested by a job are first sought in the buffer pool. Only if the blocks are determined not to be in the pool does MIIS go to the physical disk; then the block is transferred into the buffer pool so that the next (or same) job may access the block again from memory. Blocks remain in the buffer pool until overwritten. The least-recently looked-at block is the block which is overwritten first.

Locating a particular block in the buffer pool involves searching the entire buffer pool table. For this reason, MIIS segments the buffer pool into several smaller pools, each with its own table. Each such pool stores only blocks whose block numbers are equivalent when taken modulus the number of buffer pool segments. The number of pools is established when the system is first configured, and may be changed with the REC (system reconfigure) utility program.

One disk buffer is dedicated to storing map blocks (see section 13.2, Physical Storage of Globals and Programs), in order that map blocks (which are written out immediately upon being updated) do not clutter up the rest of the buffer pool.

Referencing Globals

There is a several-step process involved in accessing global information. The intent of the process is to minimize the amount of work (time) that it takes for the average job to retrieve its information, while at the same time reducing the risk of allowing a system failure to create an error in the structure of the database.

If the global reference is via one of the nine namespaces, or is embedded in a $NEXT function or $GLOBALGET function, MIIS first attempts to satisfy the reference by looking at a single block in the buffer pool. This is accomplished by saving the buffer number and block number of the last level zero (data-level) block read in by a namespace, $NEXT or $GLOBALGET reference. If the same block number is still stored in that buffer at the time of the subsequent reference, MIIS goes directly to the buffer to try and locate the required data.

If the first attempt at retrieving the global data fails (either because the level zero block is no longer in the buffer, or because the data is not in that block), MIIS tries to satisfy the global request by tracing down the required node from the top of the B-tree (i.e., from the global directory block). In this effort, MIIS will look only at blocks which are already in the buffer pool, and will not address the physical disk at all. If all the blocks from the global directory down to the level zero block that contains the required data are present in the buffer pool, MIIS satisfies the reference in this way, from local memory.

These first two steps (the single-block buffer reference and the B-tree buffer reference) take place without the job's ever "owning" the disk. In other words, a second job may be performing physical disk I/O (waiting for a block to come in from the disk) at the same time the first job is referencing blocks in the buffer pool.

If both attempts fail to satisfy the request for information, the attempt must have failed because a particular block was not present in a buffer. In this case, the job goes on the end of the disk queue in order to fetch the missing block from the physical disk. When the job gets to the front of the disk queue, it establishes ownership of the disk and requests the appropriate block. The job remains disk hung while the block is being transferred, and when the block comes in, the job finishes its logical reference immediately by scanning the new block for the number of the next block in the B-tree structure. Eventually, the level zero block is transferred into the buffer pool. When this happens, the job is placed on "queue-immediate" which is a special holding place for jobs whose physical disk references have just been satisfied.

When the job that is currently running (i.e., the job that was interrupted by the physical disk returning the level zero block) finishes its slice, the queue-immediate job releases the disk and becomes the currently running job. The job then gets a full time slice (depending on the $RESOURCE reserved word), and at the end of its slice, the job returns to the end of the CPU wait queue.

Subsequent requests by the job for global information are handled in the same manner. Since a job that is requesting physical disk may monopolize the disk by continuing to bounce back and forth between being on the wait queue, being disk hung, and being on queue-immediate, a counter is maintained to limit such a loop. The number of times that a job is allowed through this loop is controlled by $RESOURCE.

Writing Global Information

Changing (writing) global information can cause the B-tree structure to change (because a disk block must be split or because a block is removed from the structure with a KILL command), or an update to the database may take place entirely within a single block with no effect on the B-tree structure.

There are two flags associated with each block image in the buffer pool. The first, called the "high priority write switch", indicates that the block image has been changed as the result of an alteration to the B-tree structure. The second, called the "low priority write switch", indicates that the block image has been changed, but that the structure of the database has not been changed.

Because MIIS wants to minimize the risk of being interrupted (by a power failure, etc.) while having only partially reflected a structure change on the physical disk, sets of blocks which require high priority writing are written to the disk at the first opportunity the system has to obtain ownership of the disk. These changes are written in a single set of consecutive block outputs. From the time the first block in the group is written until the last block is written, the system's "high-priority-write-in-progress" switch is set. This switch is referenced by system restart/reload instructions in order to ensure that a high priority write is not interrupted.

Single-block writes (low-priority) may be thought of as being somewhat less critical, because not writing them out results only in lost data, and not in errors in the database. These outputs are written to the disk either during periods of low system activity, or when MIIS needs to store a different block in the buffer which contains the image of the altered block.

14.5 The Timesharing Executive

The timesharing executive of the MIIS operating system makes decisions based on a variety of factors. The following is an outline of actual algorithm that is used to decide what to do next:

```
IDLE:   IF <disk is not busy> THEN
                IF <high-priority write switch> THEN <run HP writer and idle>
                IF <there is a job on the disk queue> THEN
                        IF <cons switch not set> THEN <run it and idle>
                        IF <job is console job> THEN <run it and idle>
                IF <low-priority write switch> THEN <run LP writer and idle>
                IF <cons, down, halt switches> THEN <halt system>
        IF <q-immed job exists> THEN
                IF <q-immed job doesn't own disk> THEN <error - halt>
                IF <high-priority write switch> THEN <run HP writer and idle>
                ELSE <run q-immed job>
        IF <cons switch set> THEN
                IF <console job on a wait queue> THEN <run it and idle>
                ELSE <idle>
        IF <there is a job on a wait queue> THEN <run it and idle>
        ELSE <idle>
```

CHAPTER 15: Summary of Significant Utility Programs

The MIIS utility set is intended to fulfill two purposes: to make the applications programmer's life easier by providing capabilities that he would otherwise have to develop for himself; and to provide the computer operations personnel with tools to enable them to manage the computer system and to support the activities of the programmers and the users.

The following is a brief description of the most important categories of utilities. More information about each of the programs described below is contained in the MIIS Utilities Reference Manual.

Time and Date Utilities

The time and date utilities are as follows:

Name	Description
%TR@I	Converts external time to internal time (seconds since midnight).
%TR@O	Converts internal time to external time (hh:mm AM/PM).
%DT@I	Converts external dates to internal dates (number of days since March 1, 1800).
%DT@O	Converts internal dates to external dates.
$DT	Converts internal dates to external dates using $-sign space (equivalent in function to %DT@O).
$DR	Converts external dates to internal dates by checking first for "T+n" dates, and then invoking %DT@I.
%DM	Converts internal times (number of seconds since 12:00:01 AM, Jan 1, 1976) into day/date/time format (like: WED JUN 11, 1980 10:41 AM).
STD	Initializes computer's $TIME reserved word.
DLS	Changes $TIME reserved word at daylight savings time.
TIM	Manages multiple time zone table for machines configured to support it.

Program and Global Utilities

The global and program utilities are as follows:

Name	Description
%P	Prints a list of programs in the current password.
%P@ID	Interactively compiles a list of programs in the current password.
%PP	Prints a set of programs.
%PS	Searches a set of programs for a context-dependent string of characters.
%PF	Prints the first lines of a set of programs.
%PC	Copies code from one program into another program.
%PT	Copies programs from tape or on-line disk to another password or medium.
%G	Lists globals or prints a particular set of global structure.
%G@E	Edits (modify) globals.
%GN	Prints globals that may have <nil> subscripts.
%GT	Copies globals from tape, on-line disk, or off-line disk.

System Performance Utilities

The system performance utilities are as follows:

Name	Description
%SI#	System information about a particular job: devices owned, password, $QUEUE, $RESOURCE, and the currently executed program.
%SJ	%SI# information, as well as a snapshot of the code being executed by the job and the local symbol table.
%SS	%SI# information about each job that is currently signed on, sorted by the various queued or hung states that the jobs are in.
%SS@WHO	Information similar to that in %SS, sorted by job number.

%SL System loading statistics, displaying average queue lengths and which jobs are spending time in which queues. Shows which applications programs are monopolizing system resources.

%SM System measurement statistics, showing buffer pool management efficiency.

Caretaker and Error Monitor

There is a background program (named CT) that runs at all times. This program monitors the activity of the computer system, printing messages on the console terminal about significant events as they occur. Such significant events include errors in applications programs, unsuccessful attempts to sign on to the system, soft (corrected) and hard (uncorrected) disk errors, periodic lists of the users currently signed on to the computer, and messages passed to the caretaker by applications programs.

Whenever an applications program that is running in user mode is issued an error by the operating system, the job is signed off. In addition, an indication is made by the operating system to the caretaker utility that an error has occurred. The utility then takes a snapshot of the partition in which the error occurred and stores the snapshot in a global called ^%ER.

There is another utility program (called %ER) that will display the contents of this global so that programmers may ascertain what the program was doing at the time it was interrupted by the error.

Dump Program

In order to back up the system, it is necessary to make periodic copies of the disks. These copies may be made either on another disk pack, a diskette (floppy) cartridge, or a magtape. There are system "dump" utilities that will make a physical copy of a disk pack onto any of those media.

The usual technique is to make daily copies of disk packs onto off-line packs. In this way, blocks may be randomly accessed and retrieved in the event of a system failure.

Pack Check, Block Tally, Free Count and Map

There are four utilities that are concerned with making sure that the data that is stored on the disk is still intact, and that there is sufficient space available for the data to grow as necessary.

The pack check utility (called PC#) makes sure that the redundant pointers in the B-tree structure (see section 13.2, Physical Storage of Globals and Programs) are in fact redundant, indicating that the structure is sound.

The block tally utility (called BT#) counts the number of blocks that are actually used by a particular password, and breaks the utilization down by B-tree level and, for level zero blocks, by which global is stored in the block.

Both of these utilites should be run on a weekly basis in order to locate trouble areas as they occur.

The free count utility (called FC#) counts the number of free blocks left in a particular password's allocated space.

The MAP utility displays how many blocks in each map (see section 13.2, Physical Storage of Globals and Programs) are not in use.

Initial Program Load

The initial program load utility (called IPL#) leads the operator through the process of bringing up the system. The IPL# utility performs the following actions:

1) Invokes STD to set time and date,
2) Invokes FC# to count the number of free blocks,
3) Starts the caretaker program CT, and
4) Makes all devices breakable so that users may sign on.

Dollar-Space Utilities

Dollar-space contains MIIS code that may be executed without being loaded in a program buffer. The code in dollar-space is shared among all the users on a particular machine, and is loaded into dollar-space with the utility program DOL.

The following subroutines are stored in dollar-space:

Name	Description
$E	Program editor. In dollar-space so that it may operate on loaded programs.
$DT, $DR	Date conversion utilities described above. In dollar-space for speed of access.

CHAPTER 16: Appendix I - ASCII Codes

ASCII COLLATING SEQUENCE

0	NULL	26	SUB	52	4	78	N	104	h
1	SOH	27	ESCAPE	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	CS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32	SPACE	58	:	84	T	110	n
7	BELL	33	!	59	;	85	U	111	o
8	BS	34	"	60	<	86	V	112	p
9	HT	35	#	61	=	87	W	113	q
10	LINEFEED	36	$	62	>	88	X	114	r
11	VERT TAB	37	%	63	?	89	Y	115	s
12	FORMFEED	38	&	64	@	90	Z	116	t
13	RETURN	39	'	65	A	91	[	117	u
14	SO	40	(	66	B	92	BACKSLSH	118	v
15	SI	41	)	67	C	93	]	119	w
16	DLE	42	*	68	D	94	^	120	x
17	DC1	43	+	69	E	95	_	121	y
18	DC2	44	,	70	F	96	`	122	z
19	DC3	45	-	71	G	97	a	123	{
20	DC4	46	.	72	H	98	b	124	|
21	NAK	47	/	73	I	99	c	125	}
22	SYN	48	0	74	J	100	d	126	~
23	ETB	49	1	75	K	101	e	127	RUBOUT
24	CAN	50	2	76	L	102	f		
25	EM	51	3	77	M	103	g		

CHAPTER 17: Appendix II - Disk Block Structure

The following is a list of the abbreviations used in the block outlines:

SYMBOL	MEANING
O(H), O(M), O(L)	Block offset; high-order, middle, and low.
CP(H), CP(M), CP(L)	Block continuation pointer; high, middle, and low.
LEV	Block level.
LAST	The entry sequence number in the block that was last SET.
EO	Entry offset, including itself. The sum of the entry offsets equals the block offset.
RC	The repeat count, representing the number of characters the current entry repeats from the previous entry.
N(1), N(2)...	The first, second, etc., characters of the data name.
D(1), D(2)...	The first, second, etc., characters of the data value.
P(H), P(M), P(L)	Down-pointers to the next lower level of structure.
TO(1), TO(2)	Tag offsets, indicating the number of characters to the next program tag. If there is no tag on the given program line, these two bytes are not present, nor are the following TAG(i) bytes.
PN(1), PN(2)...	The first, second, etc., characters of the program name.
TAG(1), TAG(2)...	The first, second, etc., characters of a program tag.
PGM(1), PGM(2)...	The first, second, etc., characters of a program line.
TAB	Tab character (211 octal).
001, 000	Bytes with octal values 1 and 0 respectively.

DATA GENERAL

Global Pointer Block

Even Byte	Odd Byte
O(H)	O(L)
LEV	CP(H)
CP(M)	CP(L)
EO	RC
N(1)	N(2)...
001	P(H)
P(M)	P(L)

Global Level 0 Block

Even Byte	Odd Byte
O(H)	O(L)
LEV	CP(H)
CP(M)	CP(L)
EO	RC
N(1)	N(2)...
001	D(1)
D(2)...	000

Program Pointer Block

Even Byte	Odd Byte
O(H)	O(L)
LEV	CP(H)
CP(M)	CP(L)
EO	RC
PN(1)	PN(2)...
001	P(H)
P(M)	P(L)

Program Level 0 Block

Even Byte	Odd Byte
O(H)	O(L)
LEV	CP(H)
CP(M)	CP(L)
TO(1)	TO(2)
TAG(1)	TAG(2)...
TAB	PGM(1)
PGM(2)...	000

PDP-11 SERIES

Global Pointer Block

Odd Byte	Even Byte
LEV | O(H)	O(L)
CP(H)	LAST
CP(L)	CP(M)
RC	EO
N(2)...	N(1)
P(H)	001
P(L)	P(M)

Global Level 0 Block

Odd Byte	Even Byte
LEV | O(H)	O(L)
CP(H)	LAST
CP(L)	CP(M)
RC	EO
N(2)...	N(1)
D(1)	001
000	D(2)...

Program Pointer Block

Odd Byte	Even Byte
LEV | O(H)	O(L)
CP(H)	LAST
CP(L)	CP(M)
RC	EO
PN(2)...	PN(1)
P(H)	001
P(L)	P(M)

Program Level 0 Block

Odd Byte	Even Byte
LEV | O(H)	O(L)
CP(H)	LAST
CP(L)	CP(M)
TO(2)	TO(1)
TAG(2)...	TAG(1)
PGM(1)	TAB
000	PGM(2)...

IBM SERIES 1

Global Pointer Block

Even Byte	Odd Byte
LEV | O(H)	O(L)
LAST	CP(H)
CP(M)	CP(L)
EO	RC
N(1)	N(2)...
001	P(H)
P(M)	P(L)

Global Level 0 Block

Even Byte	Odd Byte
LEV | O(H)	O(L)
LAST	CP(H)
CP(M)	CP(L)
EO	RC
N(1)	N(2)...
001	D(1)
D(2)...	000

Program Pointer Block

Even Byte	Odd Byte
LEV | O(H)	O(L)
LAST	CP(H)
CP(M)	CP(L)
EO	RC
PN(1)	PN(2)...
001	P(H)
P(M)	P(L)

Program Level 0 Block

Even Byte	Odd Byte
LEV | O(H)	O(L)
LAST	CP(H)
CP(M)	CP(L)
TO(1)	TO(2)
TAG(1)	TAG(2)...
TAB	PGM(1)
PGM(2)...	000

CHAPTER 18: Appendix III - Summary of Commands

MIIS COMMANDS

```
ASSIGN nexp,nexp:alit=exp{:alit=exp}
BREAK~~
CALL pgm{@tag}{[exp,exp]}{:lexp}
CLOSE {gvar}
DO tag{[exp,exp]}{:lexp}
ELSE~~
ERASE {tag{:tag}}
FILE {pgm:X}
FOR var=exp
FOR var=nexp:nexp{:nexp}
GO~~
GOTO tag{:lexp}
HALT~~
HANG nexp
IF {lexp}
KILL {var,*var,(var,*var),var[nexp]}
LOAD {pgm}
MOVE {tag{:exp}}
NAMESPACE~~
NAMESPACE nlit{=var}
NAMESPACE var  [equivalent to N l=var]
OPEN gvar
OVERLAY pgm{@tag}{:exp}
PRINT {{tag}{:{tag}}}
QUIT~
READ var,*var,"<literal>",<carriage control>
SET var{=exp}
SET *var{=var},*var=var&var,*var=var!var,*var=var'var
SET var[nexp]{=exp}
SET $BYTE(nexp)=nexp
SET $CONSOLE(nexp)=lexp
SET $GLOBALGET[nexp]{=exp}
SET $HASH=exp  [where "exp" evaluates to four characters]
SET $PASSWORD=nexp
SET $PIECE(var<delimiter>nexp{,nexp}){=exp},
    $PIECE(var,exp,nexp{,nexp}){=exp}
SET $QUEUE=nexp
SET $RESOURCE=nexp
SET $UPDATE(var{,nexp})
SET $WORD(nexp)=nexp
TRANSFER <{nexp:}nexp,>{nexp:}nexp
UNASSIGN nexp
VARIABLE var{=exp}
WRITE {exp{:nlitLnlit},exp:nlitRnlit,exp:nlitTnlit,nexp:nlitDnlit,
      nexp:nlitRnlit.nlit,<carriage control>,*nexp}
XECUTE exp
```

MIIS FUNCTIONS
---- ---------

$ASSIGN(nexp{,alit})
$BYTE(nexp)
$CHARACTER(nexp) [In expressions only - not on LHS of SET command]
$CONSOLE(nexp) [On LHS of SET command only - not in expressions]
$DEFINE(var{,var})
$EXTRACT(exp{,nexp{,nexp}})
$FIND(exp,exp{,nexp})
$GLOBALGET(var)
$HASHCODE(exp{,nexp})
$INVERSE(exp)
$JOB(nexp)
$LENGTH(exp)
$MULTIPRECISION(nexp+nexp-nexp*nexp/nexp{,{nlit}.{nlit}})
$NEXT(var{,var})
$PIECE(exp<delimiter>nexp{,nexp}) or $PIECE(exp,exp,exp{,exp})
$RANDOM(nexp)
$TEXT(tag)
$UPDATE(var{,nexp})
$WORD(nexp)
$XCLUSIVEOR(exp) or $XCLUSIVE-OR(nexp,nexp)

MIIS RESERVED WORDS
---- -------- -----

$ASSIGNMENT
$CONSOLE
$GLOBALGET
$HASH [On LHS of SET command only - not in expressions]
$IO
$JOB
$KONSTANT
$PASSWORD
$QUEUE
$RESOURCE
$STORAGE
$TIME
$XCOORDINATE
$YCOORDINATE

Where:

Code	Meaning
alit	Alphabetic literal.
exp	Any well-formed expression.
gvar	Global variable.
lexp	Expression evaluated as TRUE/FALSE.
nexp	Expression evaluated as a number.
nlit	Numeric literal (digit).
pgm	Well-formed program name.
tag	Well-formed tag name.
var	Any MIIS variable.
LHS	Left-hand-side.

CHAPTER 19: Appendix IV - Console Switches and DDT

Section 12.1, $TIME, $CONSOLE, and $KONSTANT discusses the $CONSOLE reserved word as a means to set and reset the computer's console switches. This section will give a more comprehensive description of the implications of the various switches.

There are eight named console switches. Their decimal and octal values may be ORed together to combine the effects of several switches.

The locations and meanings of these switches are presented in the table below:

Octal	Decimal	"i" in $C(i)	Name	Abbr	Meaning
1	1	15	Dynamic Debugging Technique	DDT	Makes the console terminal into a memory location inspection and editing device.
2	2	14	Kill Background Job	KBG	Tested by convention with $C/2#2=1 to determine when background jobs should halt themselves.
4	4	13	Mailbox	MBX	Tested by convention with $C/4#2=1 to determine whether there is a system message to be printed with the utility %MX
10	8	12	Halt	HALT	Used as follows: In conjunction with CONS switch to stop all jobs (including console job) from running. In conjunction with CONS and DOWN switches to halt system.
100	64	9	Console	CONS	Used as follows: In conjunction with HALT switch to stop all jobs (including console job) from running. In conjunction with DOWN and HALT switches to halt system. By itself to stop all jobs except console job from running.

1000	512	6	Down	DOWN	Used as follows: In conjunction with CONS and HALT switches to halt system. By itself to prevent new users from signing on to the computer.
10000	4096	3	System	SYS	Must be raised in order to FILE programs in the system password.
100000	32768	0	Write	WRT	Must be raised in order to: TRANSFER a block out to disk, or SET $WORD, or SET $BYTE.

Dynamic Debugging Technique (DDT)

When the computer's console terminal is operating in DDT mode, the terminal is no longer "plugged in" to the MIIS interpreter. It is instead connected to a special interpreter that is built into the operating system to enable accessing memory locations even when MIIS itself is not operational.

This interpreter accepts octal memory addresses and contents as its input. When ready for input, the interpreter advances to a new line. Entering an octal number will cause DDT to display the current contents of the memory location corresponding to that number, and DDT will then accept further input as follows: a carriage-return will leave the contents of the memory location unchanged, while entering another octal number will modify the contents of the memory location accordingly. This process is described in more detail in the Utilities Manual.

To enter DDT mode, raise the switch corresponding to octal 1 (the right-most data switch). On computers with virtual consoles, this switch must be raised logically, according to the rules for the particular virtual console. Many MIIS assemblies will support the convention of striking CTRL/S on the console to enter DDT mode, and striking CTRL/Q on the console to leave DDT mode.

Systems Without Physical Console Switches

Another way to modify the switch register on computers without physical switches is to enter an "S" to DDT when it is expecting an octal memory address. This will cause the internal switch register to be opened for inspection and editing just as though it were a regular memory location. On such assemblies, this is the only way to raise the system HALT switch: by entering into the switch register an octal or hex number that has the decimal "8" bit set.

CHAPTER 20: Appendix V - List of Error Messages

Error	Meaning
?UNDEF	Indicates that something is not defined. There are several circumstances in which this error is issued: Undefined namespace, KILLed looping variable in a FOR statement, Undefined tag or program, Reference to non-existant reserved word.
?SYNTX	Attempt to interpret illegal MIIS syntax.
?CMMND	Attempt to use a non-existant command. Also, attempt to execute an OVERLAY command within the scope of a DO or an XECUTE.
?FUNCT	Attempt to use a non-existant function.
?STORE	Generated when a programmed modification to the local symbol table would result in symbol-table/stack overlap. Also generated when attempting to create a program of more than 2047 bytes.
?STACK	Generated when an attempt to place an entry on the user stack would result in symbol-table/stack overlap.
?BREAK	Generated when a process is interrupted by striking the BREAK key on the user's terminal. Also generated by a BREAK command embedded in a program.
?SYSTM	Generated by trying to FILE a system password program without the SYS switch, or by violating the view-buffer/write-switch locking rules. Also generated as a result of several system-level errors which must be resolved by MEDITECH. (In particular, running out of blocks in a password's allocation.)
?MAXIM	String overflow, caused by attempting to create a string longer than the maximum length. Also caused by division by zero. Also caused by attempting to UNASSIGN device 0.
?ERROR	General system-level error message. Call MEDITECH when you see this one. Also generated when an attempt is made to ERASE tags that are out of sequence.

?IOdev For instance, ?IODSK, ?IOMAG, ?IOLPT, etc. Represents an I/O error from a particular device. ?IODSK is the most serious error and should be carefully resolved.

$ASSIGN Function 8
$ASSIGNMENT Function 43
$ASSIGNMENT Reserved Word 9, 43
$BYTE Function 8, 45-46
$CHARACTER Function 8, 77
$CONSOLE Reserved Word 9, 85-86
$DEFINE Function 8, 67
$EXTRACT Function 8, 73
$FIND Function 8, 76
$G Reserved Word 9, 61, 69, 71
$GLOBALGET Function 8, 69-71
$HASH Reserved Word 103
$HASHCODE Function 8
$INVERSE Function 8, 77
$IO Reserved Word 9, 34, 88
$JOB Function 43
$JOB Reserved Word 9, 43
$KONSTANT Reserved Word 9, 86-87
$LENGTH Function 8, 72
$MULTIPRECISION Function 8, 90-91
$NEXT Function 8, 68-69
$PASSWORD Reserved Word 9, 88
$PIECE Function 8, 74-75
  SETting 75
$QUEUE Reserved Word 9, 89, 101
$RANDOM Function 8, 91
$RESOURCE Reserved Word 9, 89, 108
$STORAGE Reserved Word 9, 88
$TEXT Function 8, 10, 13
$TIME Reserved Word 9, 85
$UPDATE Function 8, 59, 63-64, 100
$WORD Function 8, 45-46
$XCLUSIVE-OR Function 79
$XCLUSIVEOR Function 8, 78
$XCOORDINATE Reserved Word 9, 30, 32, 38, 87
$YCOORDINATE Reserved Word 9, 87

%-Globals 21, 88
%-Programs and Globals 4
%-Variables 22, 53

And/Or 27
Argument Passing 22, 49-50, 53, 55
ASCII Codes 20
ASSIGN Command 29, 31, 34

Balanced Tree Data Structure, Description of 94-98
  How the B-Tree Grows 95
Bootstrap 4
BREAK Command 7, 83
Buffers, Disk 2-3, 106
  Magtape 2, 40-41
Buffers, Program 2-3, 104
  Terminal 2, 36
  View 34, 44

CALL Command 50, 52
Choosing I/O Device 33
CLOSE Command 7, 100
Collation Sequence 20
  Data Names 23
Commands, Conditionalizing 7
  Format 7
Comments 84
Communications Between Computers 78
Concatenation Operator 28, 72

Data Compaction 65
Data Conversion (non-integer to integer) 24
Data Names 20, 59
  Collation Sequence of 23
Data Storage Management 92-98
Data Values 20, 59
Data, Global 20, 107
  Local 20
  Undefined 21
DDT 4, 86, 124
Deleting Input Characters 38
Deleting Input Lines 37
Device Characteristics 35
Disk Blocks, Allocation of 97
  Description of 92
  Label Blocks 93
  Map Blocks 93
  Structure of 93, 116-119
Disk Packs, Description of 92
Diskette, Using 39
DO Command 49, 52
Dollar Space 114

Echoing Characters 37
ELSE Command 7, 56
ERASE Command 10, 12
Errors, ?BREAK 83
  ?CMMND 48
  ?IOdev 39, 45
  ?STORE 59-60
  ?SYNTX 51
  ?SYSTM 45, 98
  ?UNDEF 47-48, 50-51
  General Description 125-126
Expressions, Evaluation of 24
  Well-Formed 17

FILE Command 7, 10, 14
FOR Command 15, 52, 57-58
Functions, Format 7

Global Directory 3
Global Directory Block 95, 107
Globals, Deleting 96
  Efficiency of Storage 96-97
  Storage of 94-96
  Updating 108
GO Command 83-84
GOTO Command 47

HALT Command 7, 82
HANG Command 82

I/O Devices, Blocked 39-40, 42
  Blocked and Unblocked 35
  Choosing 33
  Controlling Characteristics of 35
  Enter Key 36
  Principal 33-34
  Unblocked 35, 37-38
IF Command 7, 55-56
IF-SWITCH 29, 50-51, 55-56, 99
Indirection 15-16, 54
Inputting Data 29
Inputting Text 37
Integer Operators 24
Inter-processor Communications 78
Interpreter 6

Job 2
JOB Command 81, 103
Jobs, Hung 102

KILL Command 7, 22, 61-62

LOAD Command 7, 10, 13, 33
Local Symbol Table 2, 20, 103
  Entry Points to 103
LRU 106

Magtape, Using 39
Map Blocks 97, 106
Maximum Length of Character Strings 20
Memory, Allocation of 2
MIIS, Allocation of Disk Space 97
  AND and OR 27
  As Control Language 6
  Atom 17
  Binary Operators 17
  Buffer Pool 106
  Caretaker Utility 112
  Collation Sequence 20
  Commands 6-7
  Commands Summary 120
  Comments 84
  Concatenation Operator 28, 72
MIIS, Constants Table 86
  Data Names 20-22
  Data Structure 21
  Data Values 20
  Direct Mode 6, 52
  Disk Block Structure 116-119
  Dollar Space 114
  Executive 109
  Expression Evaluation 24
  Expressions 17, 24-25, 27
  File Efficiency 96-97
  Floating Point Arithmetic 90
  Free Map 97
  Functions 7-8
  Functions Summary 121
  Global Storage Efficiency 65
  High Priority Writer 108-109
  Hung States 102
  I/O Device 33
  Identifier 17
  In Core Table 86
  Indirection 15-16, 54
  Integer Arithmetic 24
  Inter-job Communication 34
  Internal Dates 32, 85
  Job Priority 89
  Left-to-Right Precedence 24
  Loading 4
  Low Priority Writer 108-109
  Map Blocks 97
  Minimum and Maximum Operators 27
  Parallel Processes 81
  Partition 103
  Patching 4
  Pattern Matching 25, 72
  Principal Device 33-34
  Programs 10
  Relational Operators 17, 25
  Reserved Words 7, 9
  Reserved Words Summary 121
  Signing Off 82
  Signing On 5
  Statement 6
  Storage of Global Data 94-96
  Subroutines 49-50, 53, 55, 61
  Time Slice 89
  TRUE and FALSE 25, 27, 55
  Unary Not 25, 27
MOVE Command 10, 12

Namespace 61
NAMESPACE Command 7, 64-65

OPEN Command 63, 99-100
Operators, Arithmetic 24
  Boolean 27

Operators, Concatenation 28, 72
Negating 25, 27
Pattern Matching 25, 72
Relational 25
Outputting Data 31
OVERLAY Command 48

Packed Data 22, 59, 61, 63, 69
Parallel Processing 81
Partition 2, 20, 103
Size of 2
Password 3, 20
Password Table 4
Pattern Matching 25, 72
Percent Programs and Globals 4
Ports, Programmable and Non-programmable 5
PRINT Command 7, 10-11, 13, 33
Program 3, 10
Program Buffers 10, 104
Program Directory 3, 10, 13-14
Program Directory Block 98
Program Labels (Tags) 10
Program Line Pointer 11
Program Lines 6, 10
Program Vector 2
Programming Mode 5, 10, 52
Programs, Deleting 14
Filing 14
Loading 13
Storage of 98
PTS Utility 4

Queues 101
Disk 101, 107, 109
Free 101
Immediate 102, 108-109
Resource 34, 42, 81, 99, 102
Run 102
Setting CPU Wait 89
Wait 42, 82, 89, 101, 109
QUIT Command 49-50, 52

READ Command 29-30, 33
Recursion 61
Reserved Words, Format 7
Resource Queue 34, 42, 81, 99

Security 5
SET Command 22, 59-60, 65
Sign On Procedure 5
Stack, User 2, 80, 83
Strings 20
Strings, Maximum Length 20
STRUCTURED SET Commands 65-66, 97
Subfile Notation 22
Subroutines 49-50, 53, 55, 61
Arguments to 49-50, 53, 55
Returning Results 55
Switches, Console 109
HLT 86
System 14, 123-124
WRT 86
System Password 4
System Reload 4
SYSTEM Switch 14

Tab Character 6
Tags (Program Labels) 10
Timesharing Conflicts 98
TRANSFER Command 44
Tree Structures 21
Trees, Balanced 107

UNASSIGN Command 42
User Mode 5
Utilities 110-113
%DV Program and Global 35
%MX 123
%OPT 99
Caretaker 112
CF Program 30, 32, 35
COP and CON 87
Date 85
DEV 102
IPL 113
Patch the System 4
PW 88
PWP 88
REC 106
STD 85
SWT 86
System Backup 112
System Integrity 113

VARIABLES Command 53-54, 60-61
View Buffer 34, 44
Location of 44

Wait Queue 42, 82
WRITE Command 31, 33

XECUTE Command 80-81

1. Chronicles Global Name Changes

MIIS Global Name	MUMPS Global Name
^A.INI	^EA(INI)
^D.INI	^EDO(INI,"SYS")
^D.INI.O	^EDO(INI)
^D.INI.DCT	^ED(INI)
^EC.INI.1	^EC(INI)
^ECD.INI.DCT	^ECD(INI)
^EDB	no change
^ED.INI.DCT	^EDD(INI)
^EG(INI.DCT	^EG(INI)
^EI.INI.DCT	^EI(INI)
^EEC	no change
^EECI	no change
^EECD	no change
^EED(INI.DCT	^EED(INI
^EEG(INI.DCT	^EED(INI
^EEJ	no change
^EEI(INI.DCT	^EEI(INI
^EK.INI	^EK(INI)
^EKILL	leading "(" removed
^EML(INI.DCT	^EML(INI
^EMN	no change
^EMS	no change
^EQUEUE	no change
^ERGE	no change
^ER($I in report gen	^ERO($I
^ER.INI in error checking utility	^ERR(INI
^ES(INI)	no change
^ESCRATCH	no change
^ESGNON	no change
^EUS(code,INI.DCT	^EUS(code,INI)
^EX(INI.DCT	^EX(INI)
^EXD(INI.DCT	^EXD(INI)
^DOC(INI.1,ITM*100+i	^DOCS(INI,ITM,i
^R.INI	^ER(INI)
^R.INI.DCT	^ER1(INI)
^RG.INI	^ERG(INI)
^RLOG	no change
^SRCH.INI	^ESRCH(INI
^SUBSET	no change
^U.INI	deleted
%YEACC	%ZeYEACC
%YEAN	%ZeYEAN
%YECHI	%ZeYECHI
%YEFLAG	%ZeYEFLG
%YEKELL	%ZeYEKLL
%YEPC	%ZeYEPC
%YEPIC	%ZeYEPIC
%YEPRTBIG	%ZeYEBIG
%YEPW	%ZeYEPW

%YEQUOTE	%ZeYEQUO
%YESP	%ZeYESP
%YETT	%ZeYETT
%ZEENET	%ZeEENT
%ZEENETO	%ZeEENTO

2. System Global Name Changes

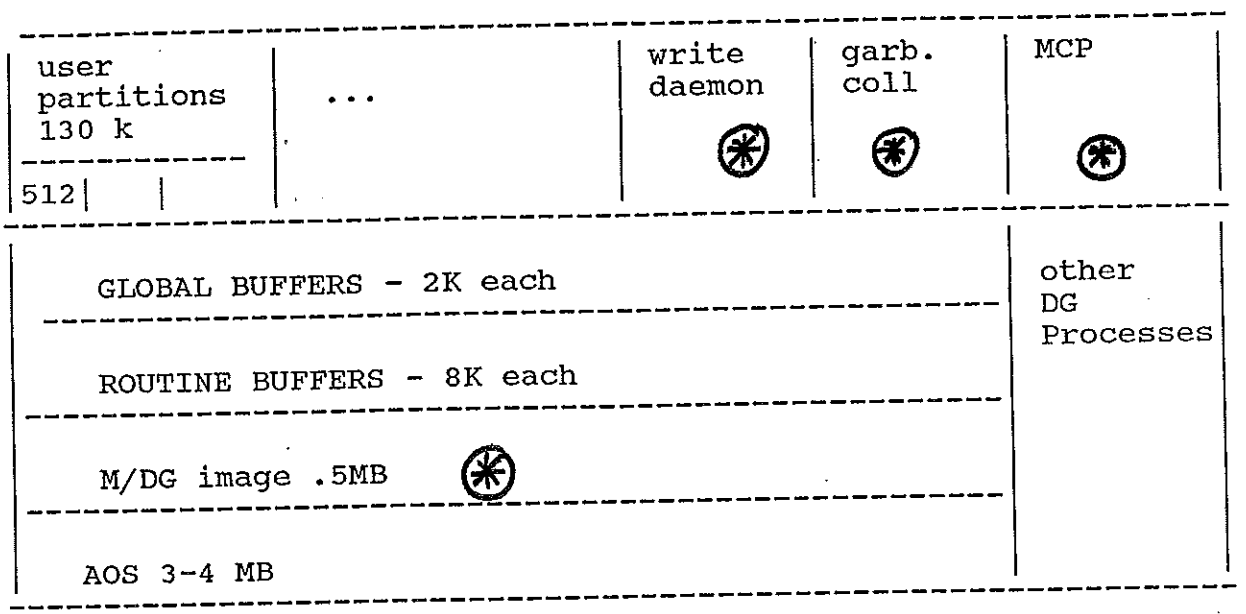
MIIS Global Name	MUMPS Global Name
DOC	DOCS
%DC	%ZeDOC
%ZBOX	%ZeUBOX
%ZDC	%ZeDOC
%ZDEVICE	%ZeUDEVN
%ZDEVICEA	%ZeUDEVA
%ZDEVO	%ZeUDEVO
%ZDVC	%ZeUDEVA
%ZELPW	%ZeULPW
%ZFTMONTH	%ZeFTMON
%ZFUNCTION	%ZeFUNC
%ZMFKEY	ZeMFKEY
%ZMFKEYDV	%ZeMFKDV
%ZMMDEF	%ZeMMDEF
%ZMMNQ	%ZeMMNQ
%ZMMNS	%ZeMMNS
%ZMMOURQID	%ZeMMQID
%ZMMR	%ZeMMR
%ZMMSQ	%ZeMMSQ
%ZMMT	%ZeMMT
%ZMMX	%ZeMMX
%ZMMX1	%ZeMMX1
%ZMONTH	%ZeMONTH
%ZMUX	%ZeMUX
%ZUDV	%ZeUDV
%ZUPR	%ZeUPR
%ZUPRF	%ZeUPRF
%ZWORDS	%ZeWORDS

3. ASAP Global Name Changes

MIIS Global Name	MUMPS Global Name
%ZEM	%ZeM (this node only, not lower nodes)
%ZEM	ZeM (all other nodes)
%ZEMACG	%ZeMACG
%ZEMAD	ZeMAD
%ZEMAIL	ZeMAIL
%ZEMBB	ZeMBB
%ZEMBLK	%ZeMBLK
%ZEMCLS	ZeMCLS
%ZEMCLU	ZeMCLU
%ZEMCNT	ZeMCNT
%ZEMCODE	%ZeMCODE
%ZEMCON	ZeMCON
%ZEMCONV	ZeMCONV
%ZEMCTS	ZeMCTS
%ZEMDEF	ZeMDEF
%ZEMDF	ZeMDF
%ZEMDOMU	ZeMDOMU
%ZEMFL	ZeMFL
%ZEMLBL	ZeMLBL
%ZEMLI	ZeMLI
%ZEMLOG	ZeMLOG
%ZEMLT	ZeMLT
%ZEMMNX	%ZeMMNX
%ZEMN	ZeMN
%ZEMNAM	%ZeMNAM
%ZEMNEWS	ZeMNEWS
%ZEMNXT	ZeMNXT
%ZEMON	%ZeUTT("NAME"
%ZEMPDF	ZeMPDF
%ZEMPKG	ZeMPKG
%ZEMSCR	ZeMSCR
%ZEMSP	ZeMSP
%ZEMTMP	ZeMTMP
%ZEMTMPI	ZeMTMPI
%ZEMTMPI9	ZeMTMPI9
%ZEMTMPJ	ZeMTMPJ
%ZEMTMPT	ZeMTMPT
%ZEMTMPV	ZeMTMPV
%ZEMUA	ZeMUA

INTERSYSTEMS MUMPS PROGRAMMING ENVIRONMENT

I. MEMORY MANAGEMENT



1. user partitions contain:
   a. symbol table
   b. routine pointers - NOT the routine code
   c. user specific information

2. memory configuration for a 64 user partition system

user partitions:	130*64 =	8.4 MB	
	AOS =	4.0 MB	
	M/DG =	.5 MB	
global buffers:	450 =	1.0 MB	(6 - 10 per user partitions)
routine buffers:	150 =	1.2 MB	(1.5 - 2 per user partitions + 20)
		15.1 MB	

3. GLOSTAT - memory performance monitoring
   a. Ratio of Global access to disk transfers should be greater than 10. If not you need more buffers.

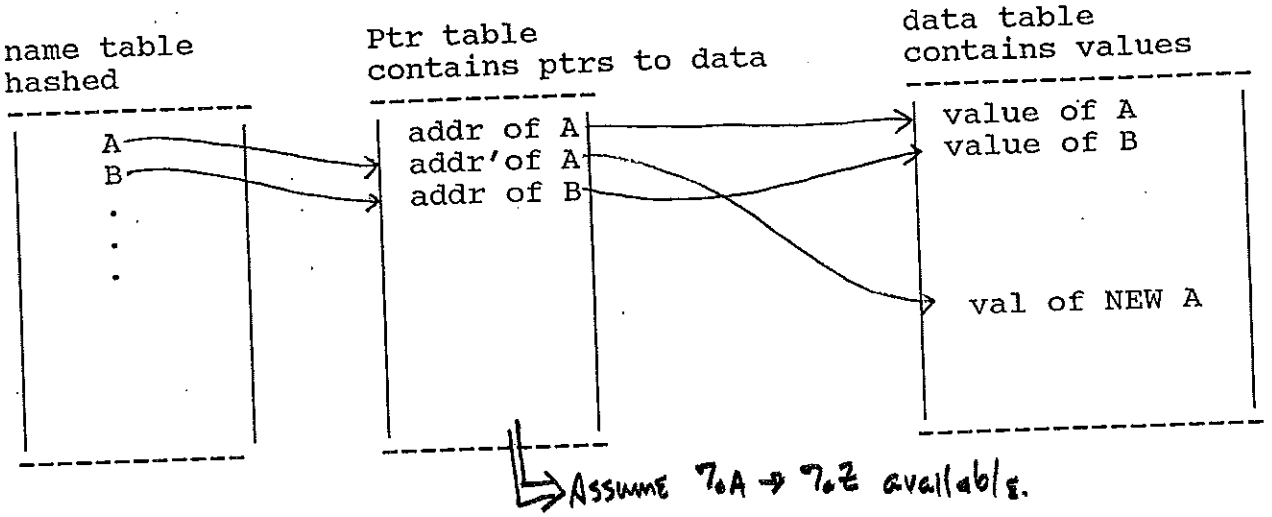
4. Write daemon functions
   a. manages buffer access asynchronously
   b. every 2 seconds writes dirty global buffers to disk

5. Garbage collector
   a. makes available killed nodes

6. Master control process
   a. initiates new MUMPS processes
   b. handles job errors

Note: M/PC, a single user system, does not have write daemon or garb. collector.

II. SYMBOL TABLE



1. dense arrays - A(1)...A(N) or close to it
   a. N must be less than 1025
   b. managed more efficently

2. max number of var names is 255 - does not include array subscripts

3. extensive NEWs are expensive New is a true local var.

III. FILE STRUCTURE

1. AOS/VS Directories - one MUMPS.DAT file per directory
   - all % globals and routines go in manager's directory

2. MUMPS.DAT structure
   a. Global Directory in blocks 2-3
      An entry for each global containing its name, implicit reference, flags for journaling and replication, an address of a Global Pointer block, and protection codes.

   b. New Global pointers
      Each global pointer is the apex of a B-tree for one global. Need about 3% of total global storage plus one block for each global

   c. Routine directory
      One block per about 225 routines

   d. Routine growth area
      About two blocks per routine

   e. Global growth area
      All data blocks for globals

   f. Every 400th block is a map block which is used to indicate free

status of the following 400 blocks

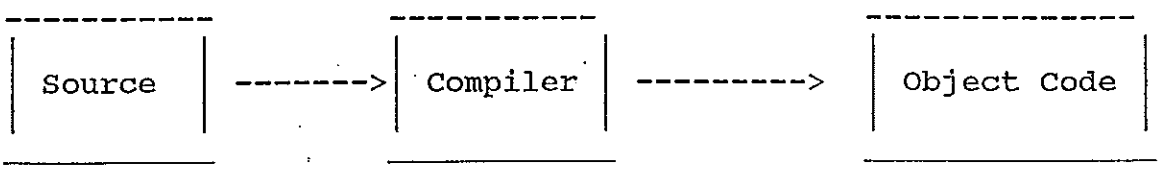
g. Example configurations:
   i. Development directory (5000 routines, 16MB for globals)
      26 maps

BLOCKS 0:2-3 Global Directory	0:4-303 Global Pointers	0:304-329 Routine Directory	0:330-5:330 Routine Growth	5:331-25:399 Global Data

   ii. Live directory (5000 routines, 100MB for globals)
      130 maps

BLOCKS 0:2-3 Global Directory	0:4-4:203 Global Pointers	4:204-4:229 Routine Directory	4:230-9:230 Routine Growth	9:231-129:399 Global Data

IV. M-code COMPILATION



MUMPS uses an incremental compiler, which means that routines are compiled line by line as needed. The object code is actually a series of simple tokens which are easy to interpret.

1. ^ROUTINE stores all routines in a given directory. This global can be manipulated like any other global.

2. ZINSERT compiles and inserts a single line into the object code module. This is executed by the editor ^% (and eventually by %ZMSP)

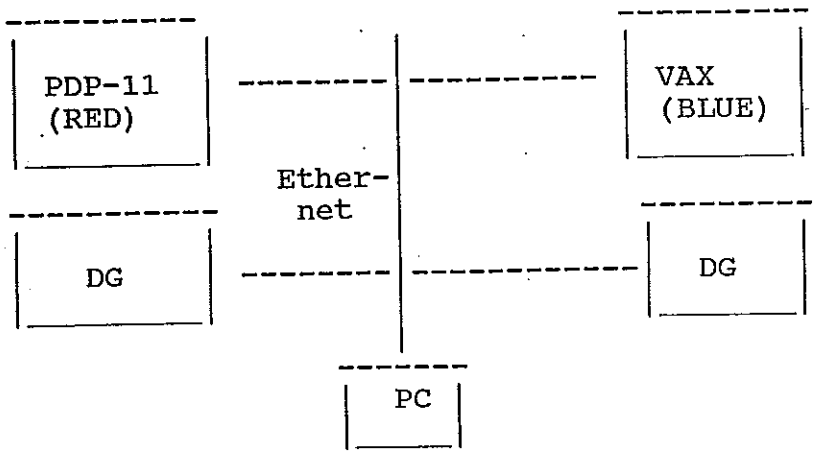
3. ZSAVE stores current routine in ^ROUTINE

4. %RCOPY can be used to recompile routines and copy routines from directory to directory.

5. Indirection is expensive because it requires a re-compilation each time it is executed.

6. $TEXT command: use double semi-colons ';;' to force the line into the object code. Otherwise, MUMPS will have to do a lookup in the source code.

V. NETWORKING



1. InterSystems MUMPS allows programs to access globals on other machines over an Ethernet. The extended global reference syntax allows you to specify a particular machine, for example

   ```
   S ^[":DIR","RED"]A(1)=1
   ```

   would set ^A(1) in directory :DIR on the computer named RED.

2. Implicit refereces can also be set up so a program can update globals on other machines without the extended references. The program GIMPLICI can be used to establish these references.

3. Replication of globals can be performed across the network. That is, every set or kill of a global can automatically be done to the same global on other machine(s). Replicating globals on the same machine in different directories is done in a similar manner. The program REPLICA can be used to create replication of globals.

4. Several different network configurations can be defined, so if one machine goes down, all implicit references and replications can be automatically transferred to/from another machine.

LAC
LPH
LCY

VI. PROTECTION

1. Protection can be set on specific globals to allow different access by Owner, Group, World or Network. Each user can be assigned a User Identification Code (UIC) which consists of two numbers (for example: 1,0). The UIC is compared against the UIC of the MUMPS.DAT file to determine if the user is attempting an Owner, Group, World or Network access. An owner is distinguished by having the same UIC as the MUMPS.DAT file. A Group member matches the MUMPS.DAT UIC in only the first number. A World member does not match in either case.

   Each global can have access control specified for each type of access. For example, global ^A may give complete access to an owner (Read,Write,Delete), but give only Read access to Group members, and no access to World or Network members.

2. Utilities
   i. MUIC may be used to set a user's UIC
   ii. MSU is used to set the MUMPS.DAT UIC
   iii. %PROTECT sets protection for specific globals
   iv. %GDISP displays protection information for globals